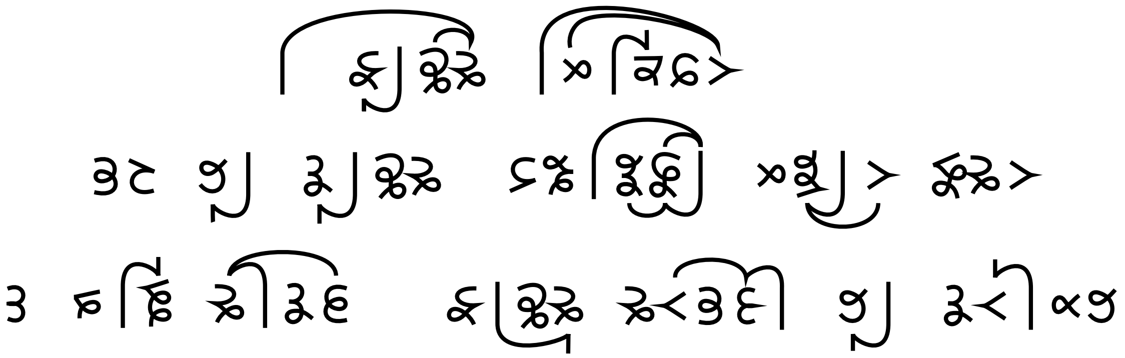
Bscript

Bipolar Binary Bitwise Script - The ultra secure human-usable cipher script



I would imagine
If you could understand morse code
a tap dancer would drive you crazy
-Mitch Hedberg

What is Bscript…….2
What is a cipher….. 2
Cipher security…….2
The Bscript cipher…….4
Symbols……. 4
Simplified symbols….. 5
Symbol Mapping…….6
Duplicate entries……. 6
Compound entries……. 8
Operators…….9
Bitwise operators…….9
Modulo operators……. 11
Mapped operators…….11
Drawing the operators…….12
Cross word operations……. 12
Input chains…….13
Operation chains….. 13
Repeated inputs…….13
Operation Density……. 14
Number of operations….. 14
Overlapping conflict……. 14
Reading and writing Bscript……. 15
Reading Bscript…….15
Writing Bscript…….15
On-line Generator tool…….15
Ultra Secure Bscript…….20
Decoding…….20
Encoding….. 20
Resources and links……. 22

What is a Bscript?

Bscript is a script optimized to cipher text, be as secure as possible and hard to crack, while still being human usable.

Bscript is optimized with the following goals

1. **Human usable –** This means you do not need a computer or device to read and write it. It must be feasible for a human to learn it and read/write it with only mental processes that are reasonable to learn and memorize.
2. **Handwritten –** It must be easy to write by hand and clear/unambiguous to read by eye.
3. **Secure –** It must be as secure as possible. For a cipher this primarily means "frequency analysis resistant"

What is a cipher?

A cipher is the most basic form of encryption. In it's simplest form it is a substitution of letters for alternative symbols with a "key" for deciphering it.

Cipher Security

The security of ciphers, codes and encryption is basically "how hard it is to crack them". A simple ciphers can be cracked with a small amount of brute force(brute force is just trying possible cipher keys until you find the key), others can be so secure not even all the computers on earth could crack them in a trillion years.

The hardest to crack encryptions require computers to use because the system of encoding and decoding is so complex that a human would take forever to just encode a single message, and a single typo would destroy the entire message.

The middle range of ciphers, where a human can still use it without any devices is what we are looking at here.

Substitution ciphers use a key where characters are swapped with their matching values in a key

eg. a=r, b=g, c=s, d=j, e=p, etc…

Polyalphabetic substitution ciphers are systems where there are many keys. You start with a specific key. Some symbols represent characters using the key, and other symbols are instructions to switch to a different a key.

Polyalphabetic ciphers are quite secure, but they require memorizing many keys, so we will not use them. You could add them on top of Bscript, but I find them hard to apply mentally without having a chart on hand. Even the "easiest" ones like the Vigenère cipher (wikipedia article) use a very simple rule to generate the keys, but is still unreasonable to use without a chart of keys or device (not saying impossible, just very burdensome)

Substitution ciphers are most commonly cracked with frequency analysis. Frequency analysis is where you analyze the frequency characters and patterns. Eg. For a simple 1-to-1 substitution cipher of common English text, I could look for 1 letter long words, these will mostly be A or I. I can then assume these symbols are either A or I, next I could look for two and three letter words that start with those characters, the highest frequency words will probably be AND, AS, IS, IN. etc…

Frequency analysis and methods to resist it are like an arms race, methods to resist frequency analysis were invented, and better frequency analysis methods were created to crack them.

Polyalphabetic ciphers can “hide” the spaces in words by encoding spaces as symbols too. This makes them hard to read without actually transcribing first (writing the deciphered message out character by character, and then reading that version)

Simple 1-to-1 substitution ciphers can’t “hide” their spaces well, even if you do substitute spaces for symbols, they are pretty quickly identified by frequency analysis no matter how hard you try,

Bscript does not attempt to “hide spaces” because it is designed to be human readable. We want to be able to decipher it without physically transcribing it, it should be feasible to read using purely internal mental processes.

A very basic method for increasing frequency analysis resistance is redundancy (having more characters than needed and switching between different options for each letter randomly)

Redundancy requires a huge index. Bscript will not rely on this, but it can be used.

I have set my total limit of “memorization units” to 100. I think this is a reasonable number.

There will be 64 Bscript characters, and few optional extra things to memorize. (operations)

So, because Bscript does not “hide spaces” or use “high redundancy” it needs a new kind of “frequency analysis immune system”.

No matter what cipher you use, no matter how secure it is, if you have enough text to analyze then frequency analysis can crack it. So the measure of frequency analysis resistance is not “can it be cracked” but instead “how much text do you need to before you can crack it”. Bscript aims to increase the required volume of text as much as possible while still being written and read mentally.

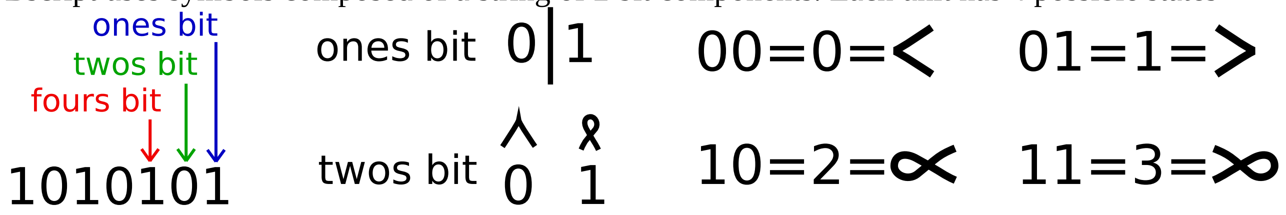
Finally there is the “Ultra secure Bscript” which first applies a baseline operation to everything to produce an encoded Bscript string. That encoded Bscript can then use all the other Bscript tools on top to further encode it.

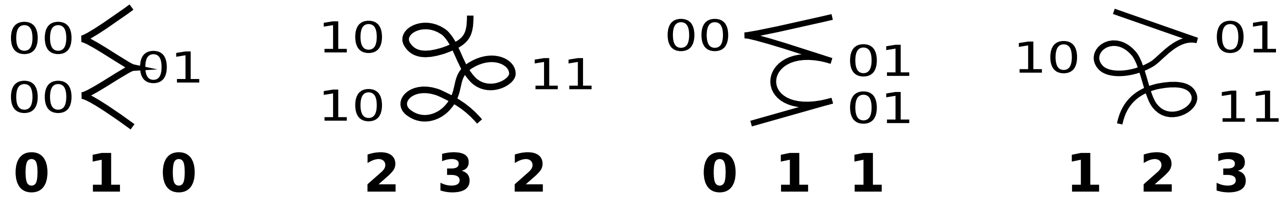
All encoding methods here are very simple. They operate on single bits, or pairs of bits. Even though there are 64 symbols, the operations are applied to small pieces of each symbol one at a time so they are quite easy to do mentally.

The Bscript cipher

Step 1 - Symbols

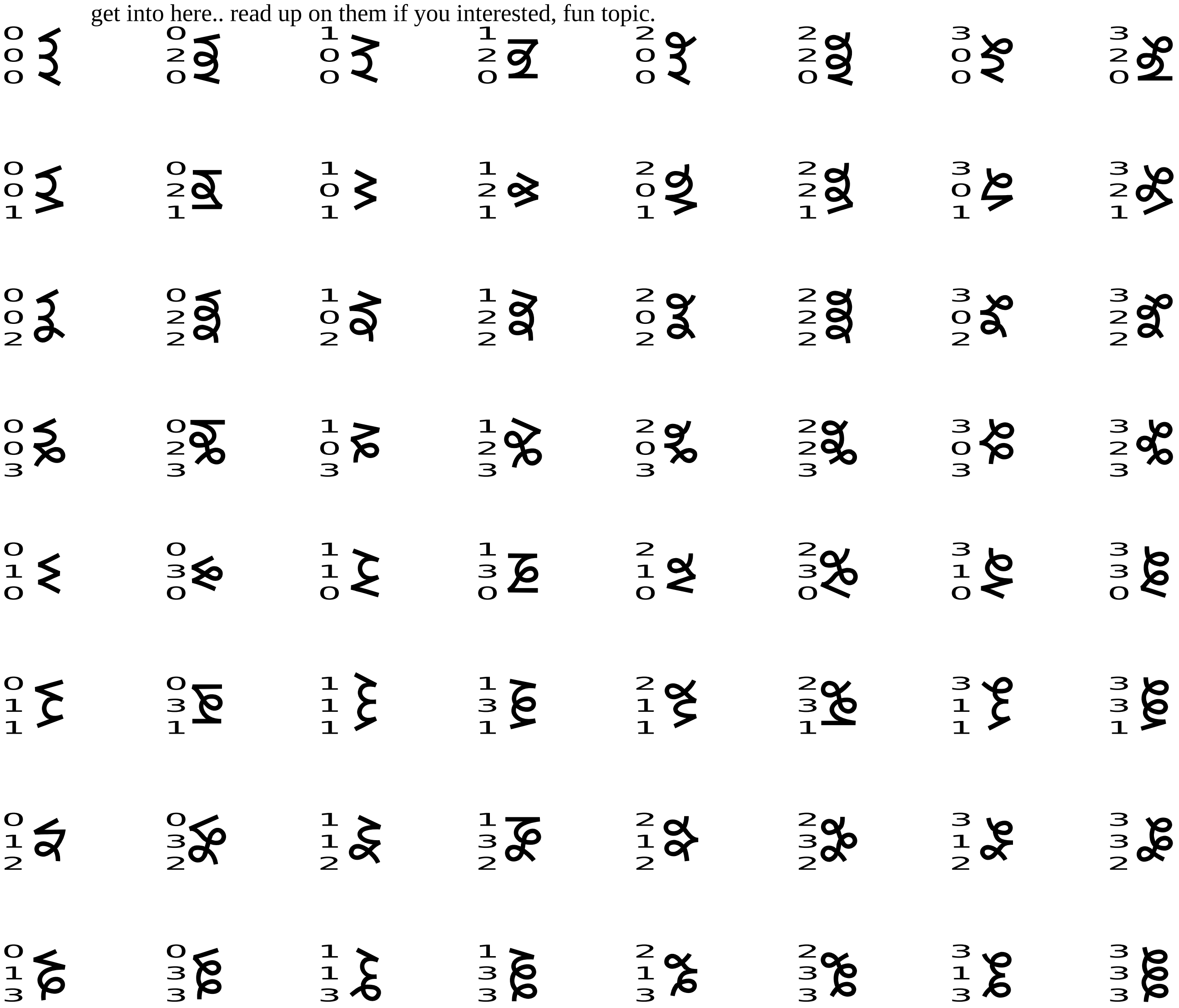
Bscript uses symbols composed of a string of 2 bit components. Each unit has 4 possible states





Bscript uses a “bipolar bit stream”. Loops are HIGH(1) and corners are LOW(0), their polarity (left/right side of the stream) like the positive negative sides of an electrical signal, represent another bit. This would be one of the types of data streams that could be used by trinary computers, polarity first bit and pulse length second bit.

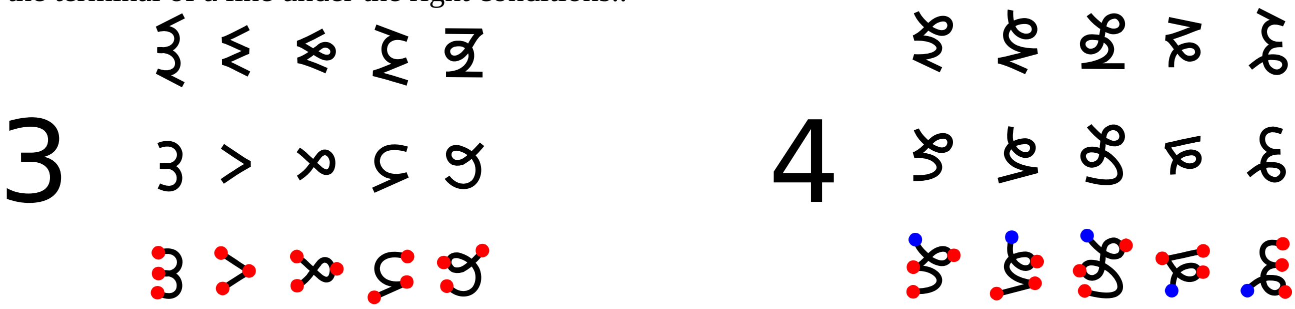
*Trinary computers had a short lived era of usage but were beat out by binary computers for reasons I won't get into here.. read up on them if you interested, fun topic.



Each bit of the 2 bit unit is also visually represented as a clear visual property, as opposed to just using 4 random symbols that don’t break into 2 visual components. This can be important depending on the “Bscript Operations” you use (keep reading, we get to these in a few pages).

Simplified Symbols

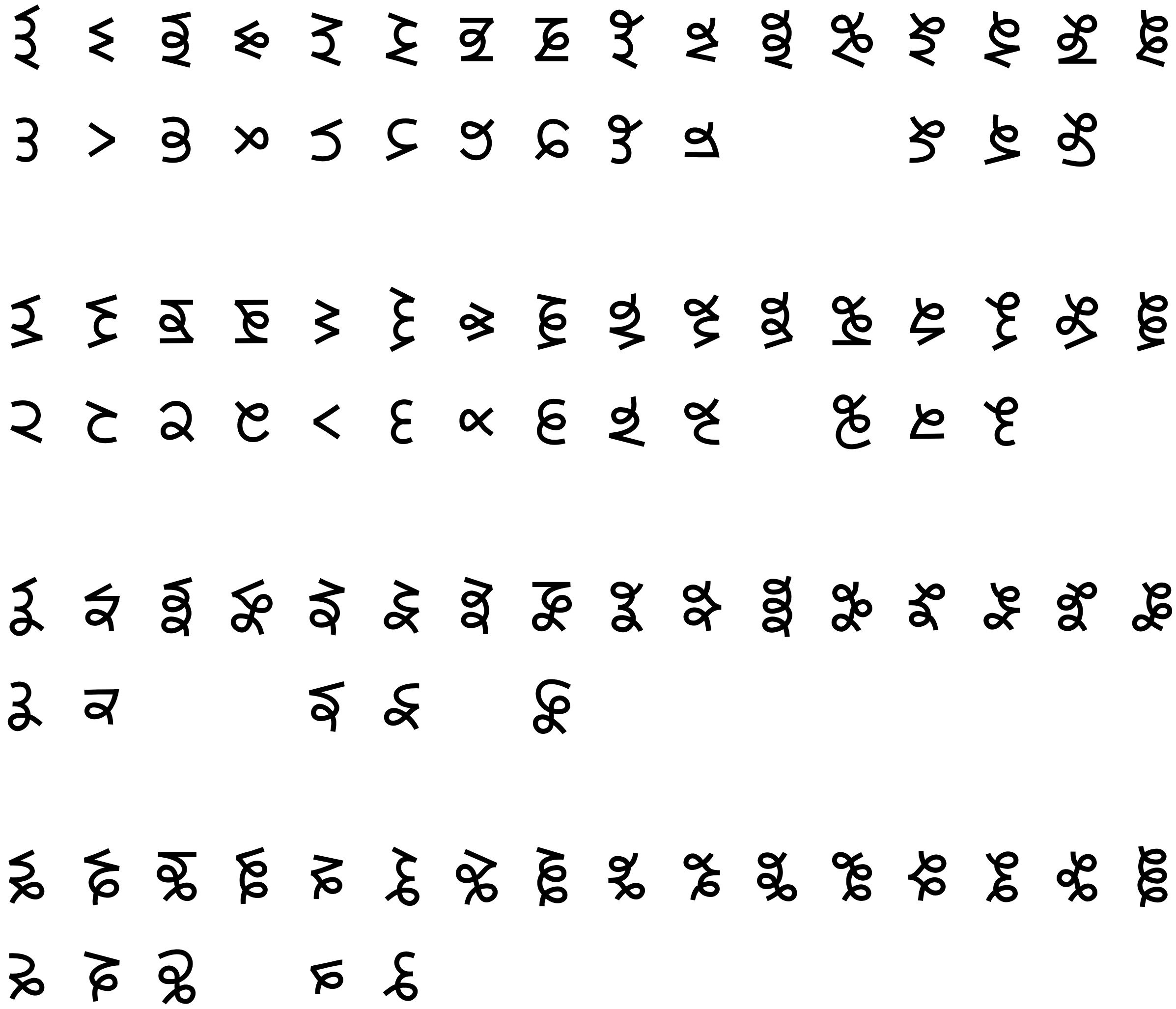
To optimize writing efficiency somehwat we can use simplified symbols. This allows us to remove some details from the top or bottom of some symbols by allowing corners (0s and 1s) to exists at the terminal of a line under the right conditions..



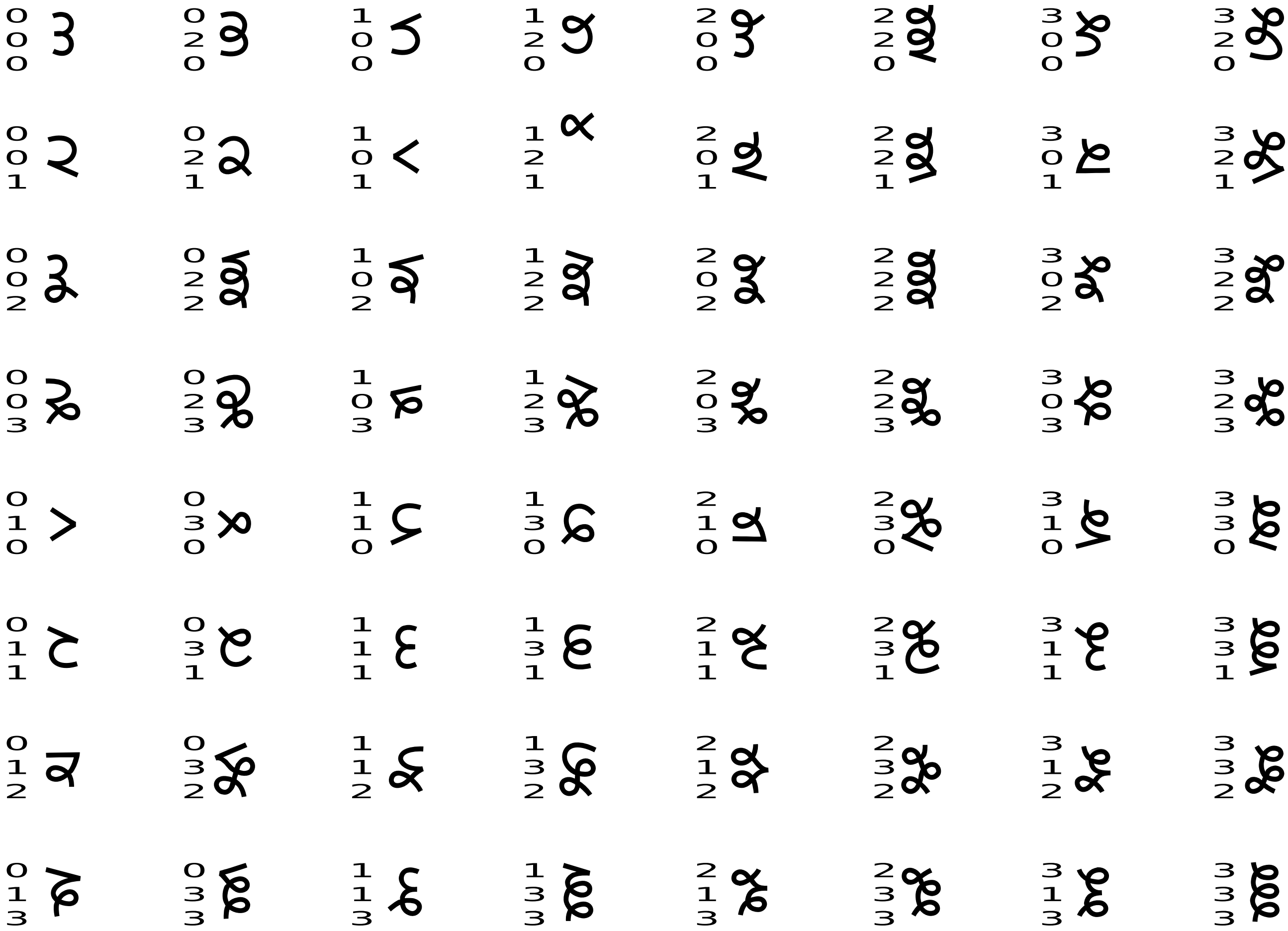
Line terminals can also represent corners.

Rules

1. Count all corners, loops, and line terminals
2. If there are 5 ignore line terminals
3. If there are 4 ignore line terminals that come out of loops
4. If there are 3 use all line terminals



Finally here is a full symbol set with the shortcut characters used when available.



Step 2 – Symbol Mapping

Now that we have a base of 64 symbols, we need to map them to characters.

For most alphabets, 32 would be enough, we have 64, but this is important for cipher security.

There are several ways to use this to resist frequency analysis.

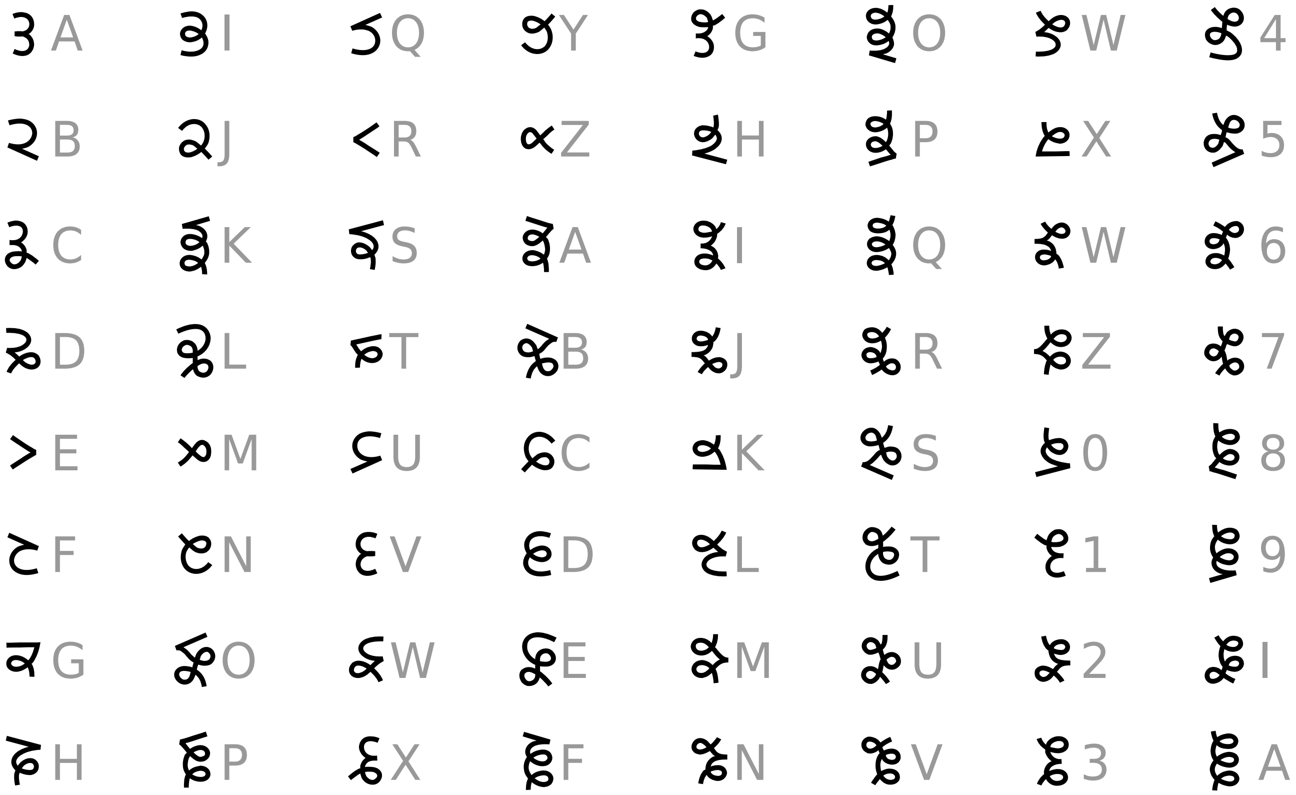
Option 1 – Duplicate entries (crack resistance : Low to Medium)
The simplest way to resist frequency analysis based attacks is to create multiple entries for letters, we could start by adding every letter twice. This would still leave some extra symbols which we could use for special characters or for more copies of frequent letters.

The most important letters to focus on are A and I because these letter can be 1 letter words. 1 letter words are very easy targets for cracking.

Multiple copies of I and A would make it harder to crack, but they would probably still be identified quickly. We could prevent this from affecting other words by only using 1 version for the 1 letter words and only using other versions for longer word. (this way identifying the 1 letter words doesn’t help you crack other words)

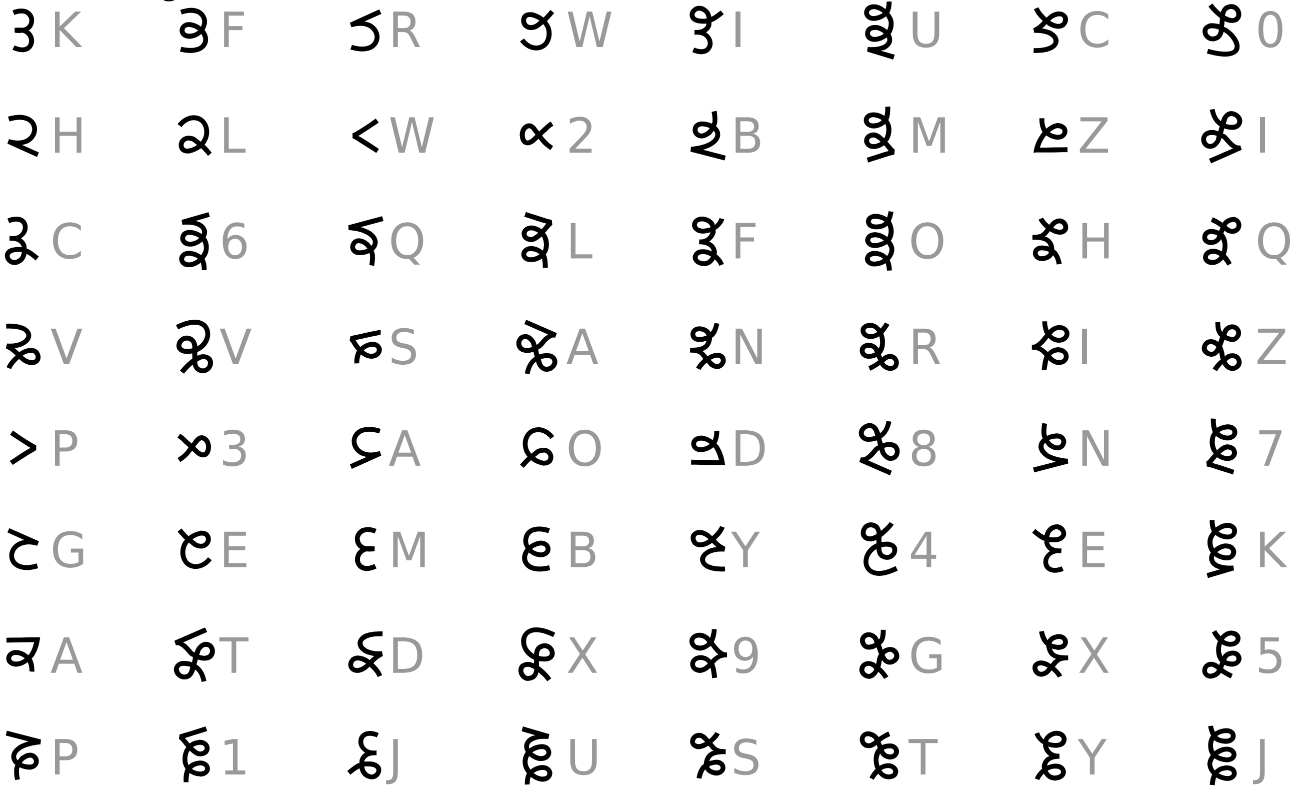
the more common the letter the more extra version should be added.

Assigning the symbols is also important, if we assign them in any pattern, such as alphabetical order, this pattern could be discovered, enabling an easy crack of our cipher even on small amounts of text.



Crack resistance : LOW

Randomizing order can increases crack resistance a bit



Crack resistance : MEDIUM

We can further increase the resistance of any cipher by allowing language modifications

- QU is very obvious because there is ALWAYS a U after a Q. There are several ways to prevent this from weakening your cipher
  eg. Allow substitutions for QU such as Q alone or KW, CW, KU, etc..

- Numbers almost always appear with other numbers and not with letters. This can be solved by allowing number letter substitutions
  eg 0=o, 1=i, 2=z, 3=E, 4=A, 5=S, 6=b, 7=T, 8=X, 9=g

- Some letters like vowels, which are common and follow patterns can have even more copies added to the index by removing extra copies of less frequent letters, or even completely removing some less common/substituted letters
  eg. remove Q and use KW, remove X and use 8, remove Z and use 2, etc..

There are of course many other ways to modify the language to increase crack resistance, but these are “layers on top of the cipher” because they can be applied to any cipher. We will not rely on these in Bscript, just be aware you can always add these to increase security.

Option 2 – Compound entries (crack resistance : Medium to Good)
An even better way to resist frequency analysis is to include some bigrams, trigrams, etc.. (bigram is a combination of 2 letters, trigram is a combo of 3 letters, etc..).

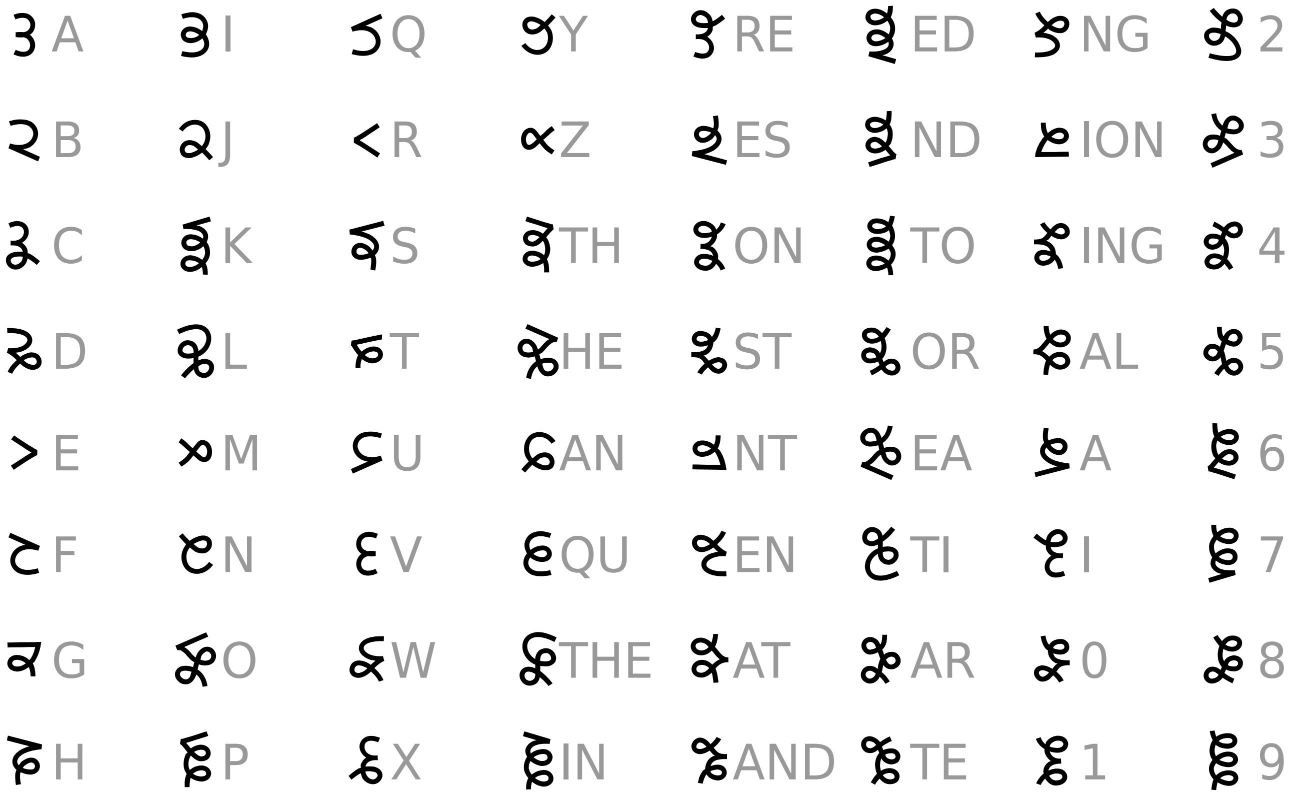
The list of common bigrams and trigrams is quite long, and you will have to choose which ones you think are best. How to choose them is tricky, there are several possible goals

- frequency : use the most common ones
  eg. TH, HE, IN,ER, AN,RE are the six most common bigrams

- compression : use the ones that will shorten words the most
  eg.THE, AND,ARE are trigrams that are common words and compose many common words

- frequency resistance : use the ones that will hide letter frequency best
  eg.QU(U always after U) and TH,SH(H often after T or S)

- language resistance : use the ones that will hide language patterns best
  eg.Use 1,2,3 instead of one,two,three, make THE,AND,OR single symbols. This creates many 1 symbol words to hide the I and A words.


A I Q Y RE ED NG 2
B J R Z ES ND ION 3
C K S TH ON TO ING 4
D L T HE ST OR AL 5
E M U AN NT EA A 6
F N V QU EN TI I 7
G O W THE AT AR 0 8
H P X IN AND TE 1 9

Crack resistance : HIGH *best to not always use compounds, randomize usage

The map is just an example of a higher resistance map, not the best possible map. To calculate the best possible map for English requires a complex analysis of the language and a lot of processing power. Best to use a new unknown map not mine.

Step 3 – Operators

This is what makes Bscript unique. Operators allow a characters to be expressed as a function of 2 other characters.

Bitwise Operators

First let’s look at using bitwise operators. Bitwise operations will already be familiar to anyone who does low level programming. These are special binary operations commonly used in the fundamental levels of computing logic.

Here are three of the most common bitwise operators AND, OR, and XOR(exclusive OR).

A bitwise operator is applied to 2 binary numbers, the operator is applied to each bit of the numbers individually.

1110101	1110101	1110101	1110101	1110101	1110101	1110101
↓	↓	↓	↓	↓	↓	↓
1101111	1101111	1101111	1100111	1100111	1100111	1100111
1 AND 1 = 1	0 AND 1 = 0	1 AND 1 = 1	0 AND 0 = 0	1 AND 0 = 0	1 AND 1 = 1	1 AND 1 = 1
1	01	101	0101	00101	100101	1100101

1110101 AND 1101111 = 1100101

Above you you see the AND operator. The and operator is is simple “if both are 1 then it output 1, otherwise, if there are any zeros, if gives 0”

Here you see how we apply the AND operator to a symbol unit in Bscript.

AND

	0	1
0	0	0
1	0	1

0	1	0	1	00	10	01	11
AND 0	AND 0	AND 1	AND 1	AND 00	AND 11	AND 11	AND 00
=0	=0	=0	=1	=00	=10	=01	=00

AND

	00	01	10	11
00	00	00	00	00
01	00	01	00	01
10	00	00	10	10
11	00	01	10	11

left = ones bit 0
∝< = X 0

loop = twos bit 1
∝∞ = 1 X

right = ones bit 1
>∞ = X 1

coner = twos bit 0
< > = 0 X

AND

	0	1	2	3
0	0	0	0	0
1	0	1	0	1
2	0	0	2	2
3	0	1	2	3

AND

	<	>	∝	∞
<	<	<	<	<
>	<	>	<	>
∝	<	<	∝	∝
∞	<	>	∝	∞

Next let’s look at a few other common bitwise operators.

OR give a 1 if any of the inputs are 1.

OR

	0	1
0	0	1
1	1	1

0	1	0	1
OR 0	OR 0	OR 1	OR 1
=0	=1	=1	=1
00	10	01	01
OR 00	OR 11	OR 10	OR 00
=00	=11	=11	=01

OR

	00	01	10	11
00	00	01	10	11
01	01	01	11	11
10	10	11	10	11
11	11	11	11	11

OR

	0	1	2	3
0	0	1	2	3
1	1	1	3	3
2	2	3	2	3
3	3	3	3	3

OR

	<	>	∝	∞
<	<	>	∝	∞
>	>	>	∞	∞
∝	∝	∞	∝	∞
∞	∞	∞	∞	∞

XOR (exclusive or) give a 1 if only one of the inputs is 1

XOR

	0	1
0	0	1
1	1	0

0	1	0	1
XOR 0	XOR 0	XOR 1	XOR 1
=0	=1	=1	=0
00	10	01	01
XOR 00	XOR 11	XOR 10	XOR 11
=00	=01	=11	=10

XOR

	00	01	10	11
00	00	01	10	11
01	01	00	11	10
10	10	11	00	01
11	11	10	01	00

XOR

	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

XOR

	<	>	∝	∞
<	<	>	∝	∞
>	>	<	∞	∝
∝	∝	∞	<	>
∞	∞	∝	>	<

As you can see some operators like AND/OR don’t have “balanced input/output”. AND returns mostly 0, while OR return mostly 1.

eg. 2 bit AND returns 9x0, 3x1,3x2,1x4

XOR is balanced, the output are evenly balanced between all possible values.

eg. 2 bit XOR returns 4x0,4x1,4x2,4x3

The advantage of bitwise operators is that they are based on simple rules, you do not need to memorize the 4x4 grid of output values.

Modulo Operators
A Modulo is simply put “remainder if divided by and we do not allow a decimal point”.

The answer to “X modulo Y” will always be less than Y

$\begin{array}{r}1\\ 1\overline{)1}\end{array}$ 1/1=1 remainder 0 $\begin{array}{r}1\\ 2\overline{)2}\end{array}$ 2/2=1 remainder 0 $\begin{array}{r}0\\ 1\overline{)2}\end{array}$ 1/2=0 remainder 1

$\begin{array}{r}0\\ 1\overline{)3}\end{array}$ 1/3=0 remainder 1 $\begin{array}{r}0\\ 2\overline{)3}\end{array}$ 2/3=0 remainder 2 $\begin{array}{r}1\\ 3\overline{)3}\end{array}$ 3/3=1 remainder 0

Mod 2 values: 0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1

Mod 3 values: 0,1,2,0,1,2,0,1,2,0,1,2,0,1,2,0

Mod 4 values: 0,1,2,3,0,1,2,3,0,1,2,3,0,1,2,3

Mod 5 values: 0,1,2,3,4,0,1,2,3,4,0,1,2,3,4,0

X mod Y =

Y

X	2	3	4	5	6	7
0	0	0	0	0	0	0
1	1	1	1	1	1	1
2	0	2	2	2	2	2
3	1	0	3	3	3	3
4	0	1	0	4	4	4
5	1	2	1	0	5	5
6	0	0	2	1	0	6

So for Bscript we want Modulo 4, this way no matter what math operation we use our answer will be between 0-3

This allows use to create some more possible operators

Addition Modulo 4

	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

	<	>	∝	ꭓ
<	<	>	∝	ꭓ
>	>	∝	ꭓ	<
∝	∝	ꭓ	<	>
ꭓ	ꭓ	<	>	∝

Absolute Subtraction Modulo 4
mod 4 (abs (x-y))

Y

X	0	1	2	3
0	0	1	2	3
1	1	0	1	2
2	2	1	0	1
3	3	2	1	0

Y

X	<	>	∝	ꭓ
<	<	>	∝	ꭓ
>	>	<	>	∝
∝	∝	>	<	>
ꭓ	ꭓ	∝	>	<

We can of course create many types of operations this way.
eg. mod 4 (X * Y), mod 4 (X+Y+1), mod 4 ((X/Y)*12), etc...

Modulo operators do not require that you memorize the entire table 16 output values.

Mapped Operators

Finally we can create custom operator maps that do not follow a specific formula.

The downside is that a map like this needs to be memorized.

The upside is that it is harder to crack.

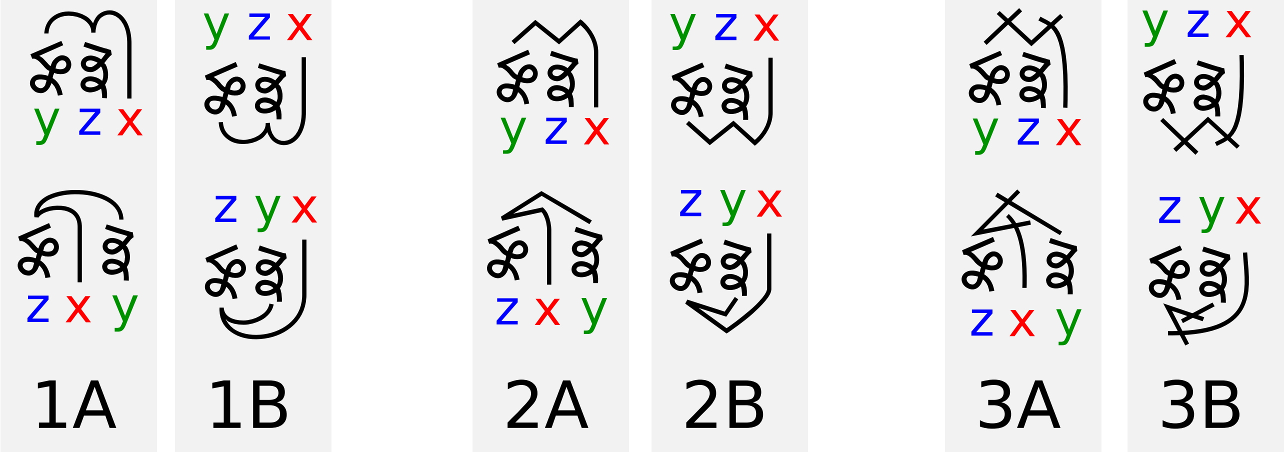
	<	>	∝	ꭓ
<	>	∝	∝	ꭓ
>	<	>	∝	<
∝	ꭓ	<	>	ꭓ
ꭓ	ꭓ	<	>	∝

Drawing Operations

Operations are drawn with simple lines.

Here are 3 operator drawing styles, each can be on top or on bottom, giving 6 possible operators.

x = y Operator z



You can of course create even more operators. As long as they are visually distinct you can have as many operators as you want.

Operators are executed like this.

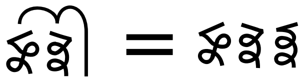
OP 1A = AND

left = ones bit 0
∝< = X 0

right = ones bit 1
>∞ = X 1

coner = twos bit 0
< > = 0 X

loop = twos bit 1
∝ ∞ = 1 X

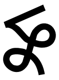


AND

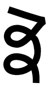
	00	01	10	11
00	00	00	00	00
01	00	01	00	01
10	00	00	10	10
11	00	01	10	11

	0	1	2	3
0	0	0	0	0
1	0	1	0	1
2	0	0	2	2
3	0	1	2	3

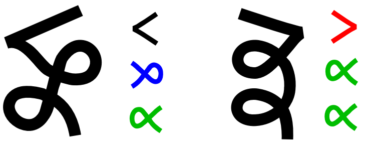
	<	>	∝	∞
<	<	<	<	<
>	<	>	<	>
∝	<	<	∝	∝
∞	<	>	∝	∞

corner left = 00  loop right = 11
loop left = 10

00 , 11 , 10 0,3,2

loop left = 10  corner right = 01
loop left = 10

01 , 10 , 10 1,2,2

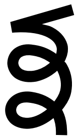


00 AND 01 = 00
0 AND 1 = 0
< AND > = <

11 AND 10 = 10
3 AND 2 = 2
∞ AND ∝ = ∝

10 AND 10 = 10
2 AND 2 = 2
∝ AND ∝ = ∝

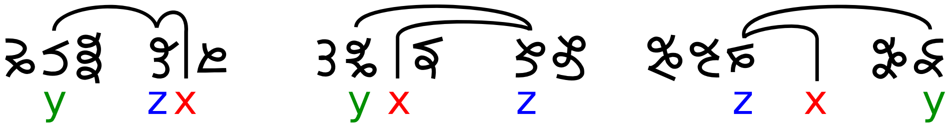
< AND > = <
∞ AND ∝ = ∝
∝ AND ∝ = ∝



Cross word operations

Operators can be executed across words

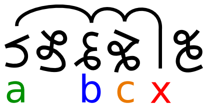
x = y Operator z



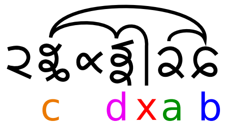
Using operators across words can be very valuable because it allows you to conceal short words which would be otherwise exposed to frequency analysis.

Input chains
Operators can also have longer chains of inputs.

$x = (a \text{ Operator } b) \text{ Operator } c$

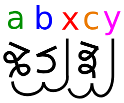


$x = ((a \text{ Operator } b) \text{ Operator } c) \text{ Operator } d$

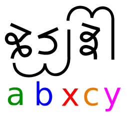


Chaining Inputs makes them more burdensome to read. I prefer not to chain my inputs because with a little practice reading single operators (2 in 1 out) can become a rather quick reflex, longer chains requires a lot more time and thought because you have to create a “mental buffer” between operations and it can easily lead to mistakes(but I suppose this is just a matter of practice and skill).

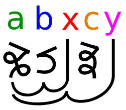
Operation Chains
The output of one operation can be used as an input for another operation.



$x = (a \text{ OP } b)$
$y = (x \text{ OP } c)$



$x = (a \text{ OP-1B } b)$
$y = (x \text{ OP-1A } c)$

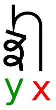


$x = (a \text{ OP } b)$
$y = (a \text{ OP } x)$

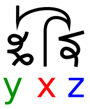
Operation chains make it very hard to crack the operations as well as the cipher in general.. If trying to crack a Bscript cipher that uses a lot of operations then cracking the operation is very important. To crack the operation frequency analysis will require a very large sample of text, operation chains will dramatically increases the required quantity of text and make cracking operations even harder.

*cracking the operation means discovering how the operation works. It is possible to crack an operation without cracking the cipher, or vice-versa. Operation chains make it harder to crack both the cipher and the operations.

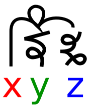
Repeated Input
An input can be used multiple times in a single operation



$x = (y \text{ OP } y)$



$x = (y \text{ OP } y) \text{ OP } z$



$x = (z \text{ OP } y) \text{ OP } y$

Adding a vertical line to the beginning of an operation can indicate that input is repeated., and a loop ca be added in the later inputs as if it “bounces and falls back onto the same input again”.

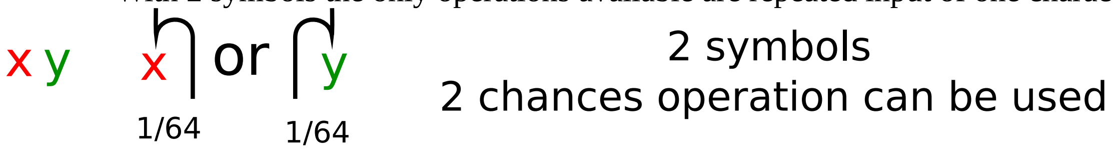
Operation Density

So how many operations can we expect to be able to used in a given section of text?

This is a tricky question. It mainly depends on 2 things

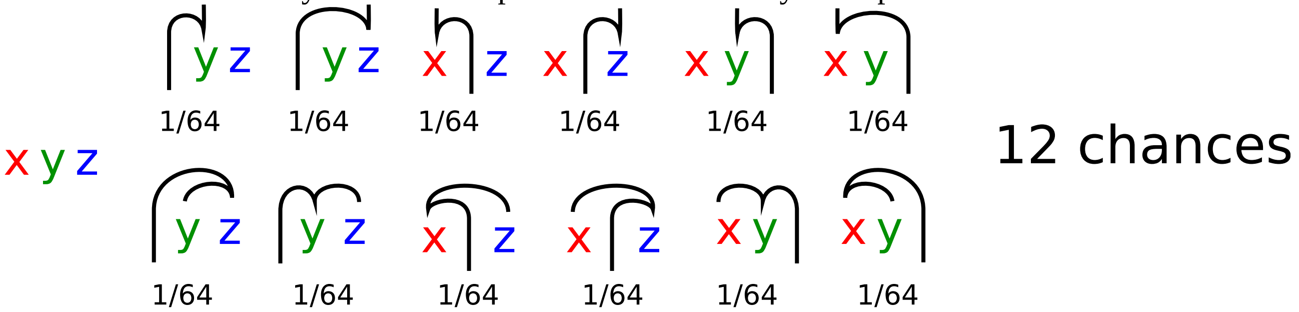
1. **Number of operation types**

With 2 symbols the only operations available are repeated input of one character.



2 operations gives 4 chances, with 4 operations there are 8 chances etc...

With 3 symbols and 1 operation we have many more possibilities.



2 operations gives 24 chances for a match, 4 operations gives 48 chances

*but wait, that means 6 operations gives 72/68, that is more than 100% how is that possible?…

answer : this is a simplified way of looking it at. There would be 72 possible operations to run, each has an output, many outputs would be the same. The accurate way to calculate the probability is

(63/64) is the probability of one operation NOT giving the desired symbol

“probability of NO match” to the power of “number of chances” = probability of NO match

1 minus probability of NO match = probability of match

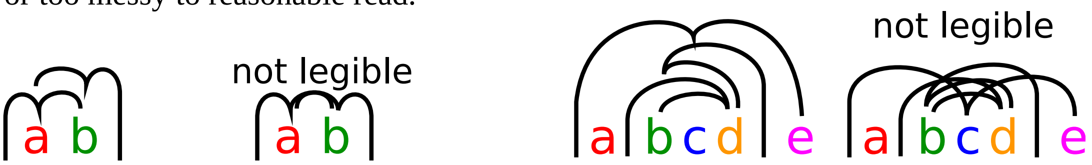
1- ((63/64)^72)   = 68%

***BUT!!!!** These probability calculations are based on randomness. If you choose your operation and symbol map carefully you can get do better than random chance by using language patterns to your advantage.

eg. there is always a U after Q, so if you make “Q OP Q = U” you will beat random chance.

2. **Overlapping conflict**

Overlapping operations can be difficult. There are situation where it might still be legible if the operations overlap, but more often they need to be vertically stacked or they become unclear or too messy to reasonable read.



Stacking can be used within reason, but if you stack to many layers to high it eventually becomes hard to identify which letters below are being pointed to.

Reading & Writing Bscript

Reading Bscript requires some practice but is quite reasonable. Reading symbols just requires learning the symbol map. Reading operations does take a bit more, but it with practice it can become a reflex. Writing Bscript symbols is quite easy. Writing operations can be tricky, specifically finding the best operations to use is the tricky part.

Reading Bscript

There are 2 key elements to reading Bscript

1. Symbols – This is a simple matter of learning the symbol for letter substitutions
2. Operations – This is a bit more complex, and there are 2 approaches
   a) memorize the 4x4 operation table
   b) learn to apply the operation logic (only applicable to bitwise and math based operators)

Writing Bscript

While writing the individual symbols is straightforward, writing operations is trickier.

Suppose you have some text you want to write. Figuring out just where you can substitute a symbols with an operations will take some time, especially if you are not yet well practiced in your symbol map and operations.

To help with writing and accelerate learning there is a Bscript online generator tool that can find valid operations.

On-line Generator tool

There is an on-line Bscript generator tool at http://www.dscript.org/bscript

This tool can be used to help generate Bscript.

Some notes about the generator

- It is primarily meant as a way to design and practice a Bscript maps and operations
- It can be used to help generate Bscript vector graphics
- It can only handle 2 operations at a time
- It only searches for operations within words, no cross word operations
- It only searches for operations within the first 10 letters of a word

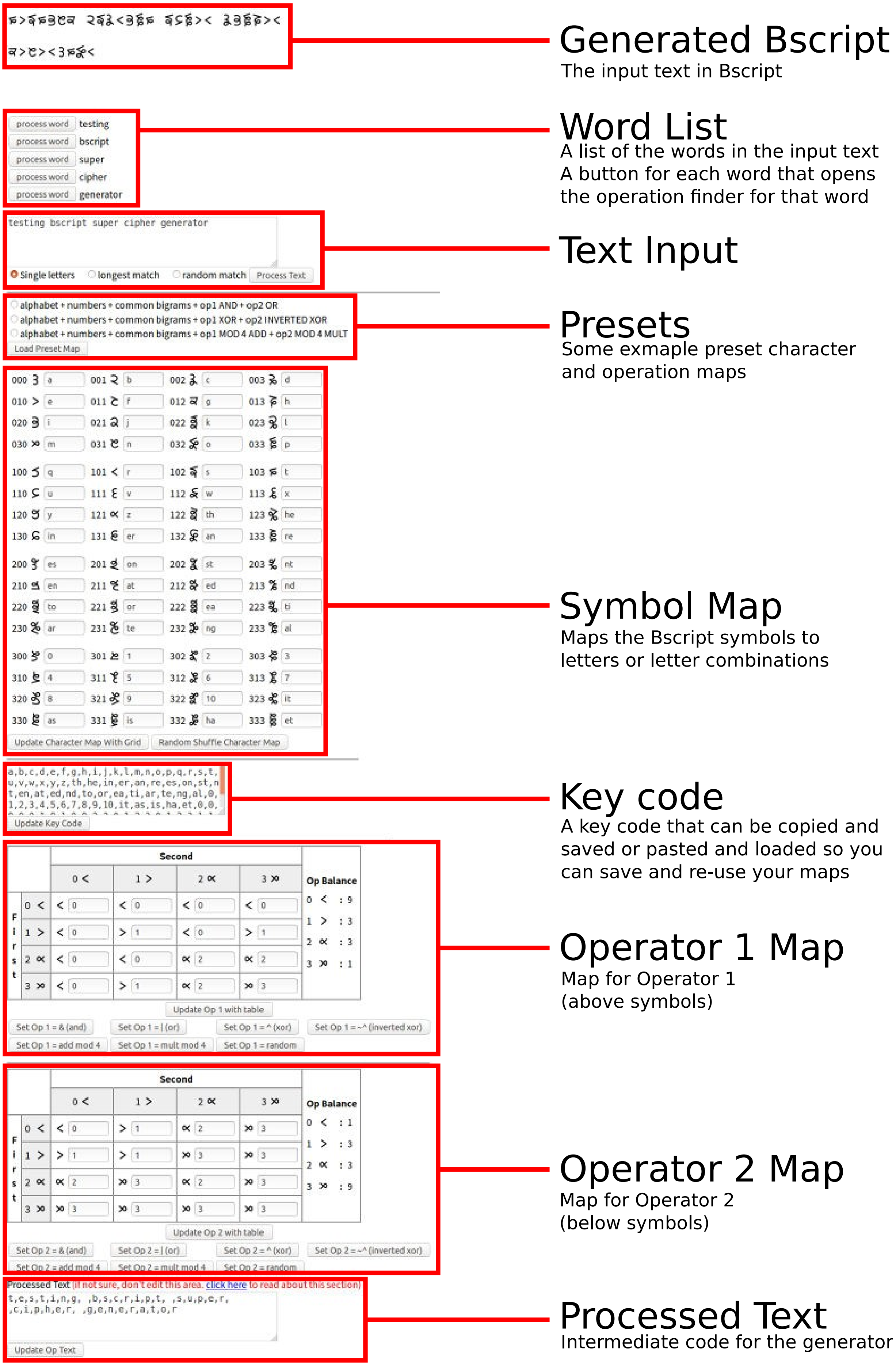
These limitations are there to prevent it from overloading the web server. Searching for combinations in long strings can take a lot of processing time.

The source code is available if you want to run it yourself and remove the limits http://www.dscript.org/bscript.zip

If you want to do cross word operations just write the words without spaces.

If you want to search for operations you can consult the operation calculator tool at http://www.dscript.org/bscript/opcalc.php. Simply copy your key code from the main generator tool and use it into the key code input on the operation calculator, then enter the value of a symbol (eg “a”, “s” “ing”, etc..) and if the character is in your map it will list all possible operation that produce it as the output.

On-line Generator Tool (http://www.dscript.org/bscript)



Generated Bscript

The input text in Bscript

Word List

A list of the words in the input text
A button for each word that opens
the operation finder for that word

process word	testing
process word	bscript
process word	super
process word	cipher
process word	generator

Text Input

```
testing bscript super cipher generator
```

● Single letters ○ longest match ○ random match Process Text

Presets

Some exmaple preset character
and operation maps

○ alphabet + numbers + common bigrams + op1 AND + op2 OR
○ alphabet + numbers + common bigrams + op1 XOR + op2 INVERTED XOR
○ alphabet + numbers + common bigrams + op1 MOD 4 ADD + op2 MOD 4 MULT

Load Preset Map

Symbol Map

Maps the Bscript symbols to
letters or letter combinations

000	a	001	b	002	c	003	d
010	e	011	f	012	g	013	h
020	i	021	j	022	k	023	l
030	m	031	n	032	o	033	p
100	q	101	r	102	s	103	t
110	u	111	v	112	w	113	x
120	y	121	z	122	th	123	he
130	in	131	er	132	an	133	re
200	es	201	on	202	st	203	nt
210	en	211	at	212	ed	213	nd
220	to	221	or	222	ea	223	ti
230	ar	231	te	232	ng	233	al
300	0	301	1	302	2	303	3
310	4	311	5	312	6	313	7
320	8	321	9	322	10	323	it
330	as	331	is	332	ha	333	et

Update Character Map With Grid Random Shuffle Character Map

Key code

A key code that can be copied and
saved or pasted and loaded so you
can save and re-use your maps

```
a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,
u,v,w,x,y,z,th,he,in,er,an,re,es,on,st,n
t,en,at,ed,nd,to,or,ea,ti,ar,te,ng,al,0,
1,2,3,4,5,6,7,8,9,10,it,as,is,ha,et,0,0,
```

Update Key Code

Operator 1 Map

Map for Operator 1
(above symbols)

First	Second 0	Second 1	Second 2	Second 3
0	0	0	0	0
1	0	1	0	1
2	0	0	2	2
3	0	1	2	3

Op Balance
0 : 9
1 : 3
2 : 3
3 : 1

Update Op 1 with table

Set Op 1 = & (and) Set Op 1 = | (or) Set Op 1 = ^ (xor) Set Op 1 = ~^ (inverted xor)
Set Op 1 = add mod 4 Set Op 1 = mult mod 4 Set Op 1 = random

Operator 2 Map

Map for Operator 2
(below symbols)

First	Second 0	Second 1	Second 2	Second 3
0	0	1	2	3
1	1	1	3	3
2	2	3	2	3
3	3	3	3	3

Op Balance
0 : 1
1 : 3
2 : 3
3 : 9

Update Op 2 with table

Set Op 2 = & (and) Set Op 2 = | (or) Set Op 2 = ^ (xor) Set Op 2 = ~^ (inverted xor)
Set Op 2 = add mod 4 Set Op 2 = mult mod 4 Set Op 2 = random

Processed Text

Intermediate code for the generator

Processed Text (if not sure, don't edit this area. click here to read about this section)

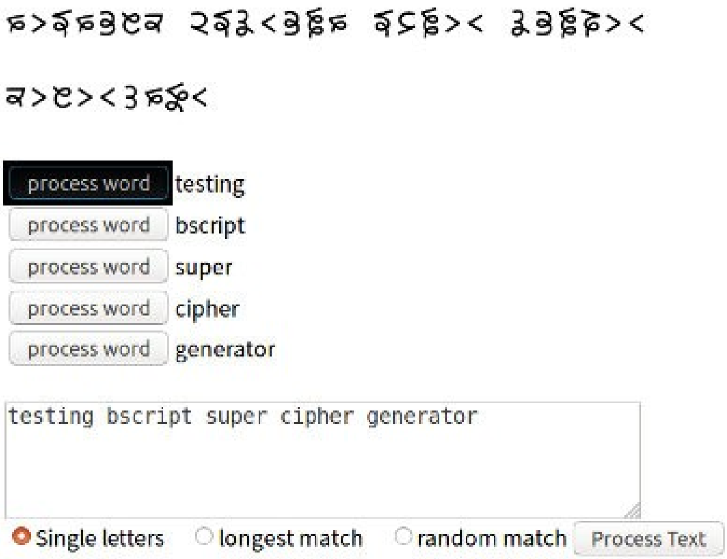
```
t,e,s,t,i,n,g, ,b,s,c,r,i,p,t, ,s,u,p,e,r,
,c,i,p,h,e,r, ,g,e,n,e,r,a,t,o,r
```

Update Op Text

Generated Bscript
Vector graphic output of the input text. SVG format, easy to modify adjust and resize. (you may need to click “save as”. Copying it may not preserve the vector format)

Word List
A list of the words from the input text.
Next to each word is a button “process text” which will bring up a list of possible operations.

You can click on the button next to a word…

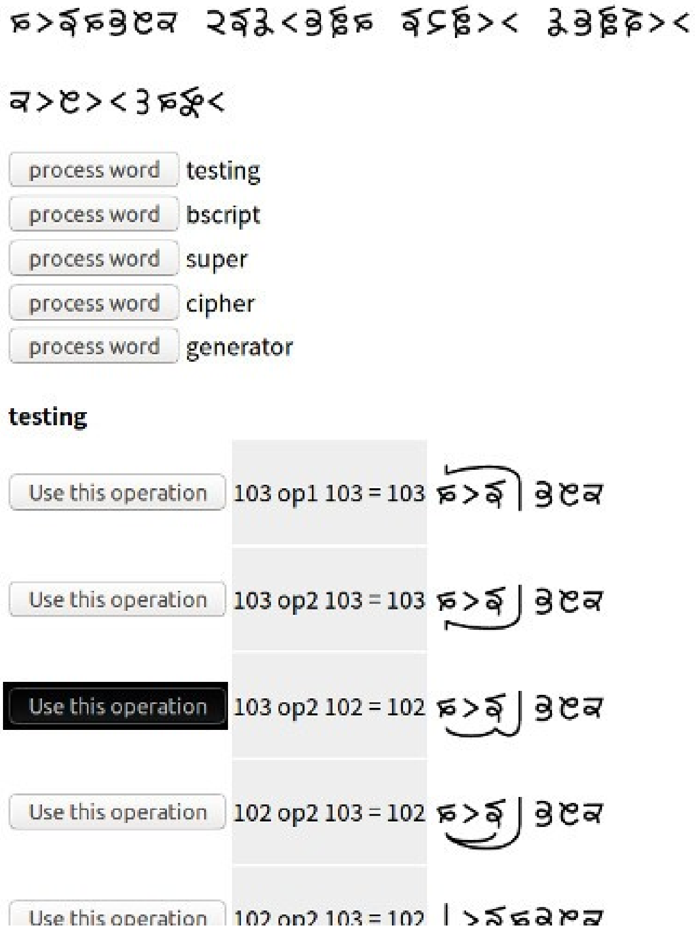


process word testing
process word bscript
process word super
process word cipher
process word generator

```
testing bscript super cipher generator
```

● Single letters ○ longest match ○ random match

Then a list of of possible operations will appear, if you click on an operation it will use it…



process word testing
process word bscript
process word super
process word cipher
process word generator

testing

Use this operation 103 op1 103 = 103

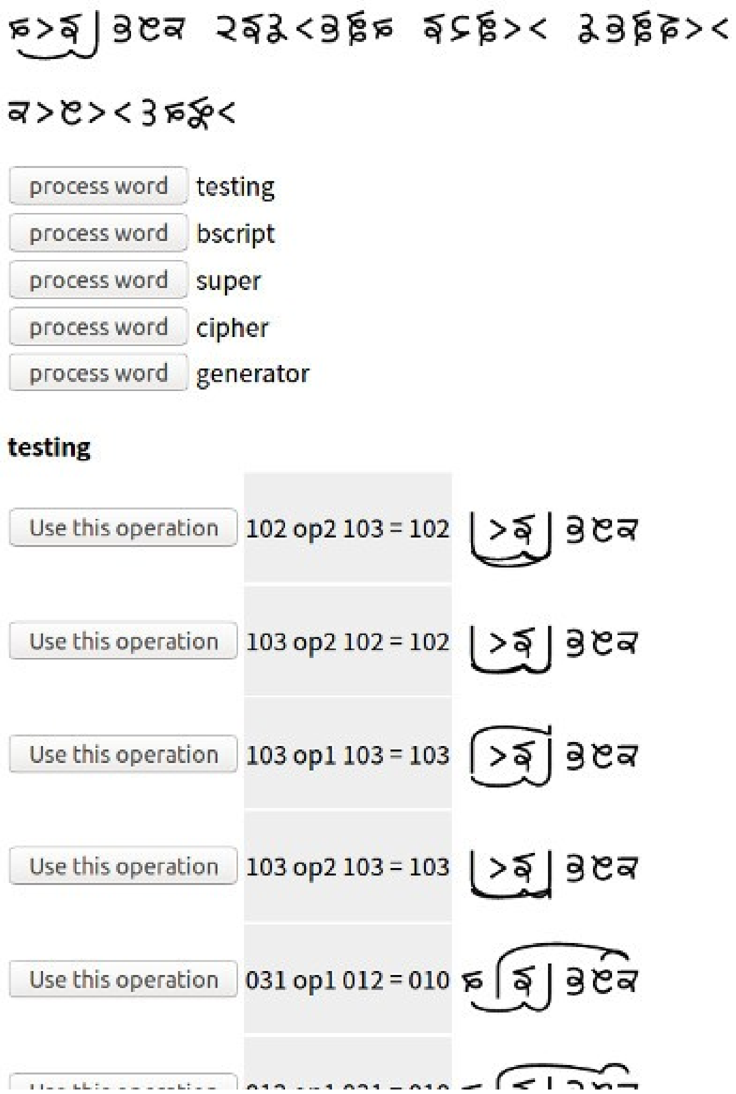
Use this operation 103 op2 103 = 103

Use this operation 103 op2 102 = 102

Use this operation 102 op2 103 = 102

Use this operation 102 op2 103 = 102

You can add more than one operation.



process word testing

process word bscript

process word super

process word cipher

process word generator

testing

Use this operation 102 op2 103 = 102

Use this operation 103 op2 102 = 102

Use this operation 103 op1 103 = 103

Use this operation 103 op2 103 = 103

Use this operation 031 op1 012 = 010

This will show all possible 2 input operations on the first 10 letters of the word.
The generator does not stack operations, so you must do that yourself
Also some operations are are mutually exclusive or create loops
A loop is where 2 or more operations cannot be solved because they rely on inputs from each other.
The third option in the list above is an example of an unsolvable operation loop.

Text Input
Here you can input the text you want to generate
When the generator goes through the text assigning symbols sometimes there are multiple options.
You can set it to either choose the first one in the list, the longest match, or a random match.

Presets
You can load from a few preset maps. The included maps are just starting points for practice.

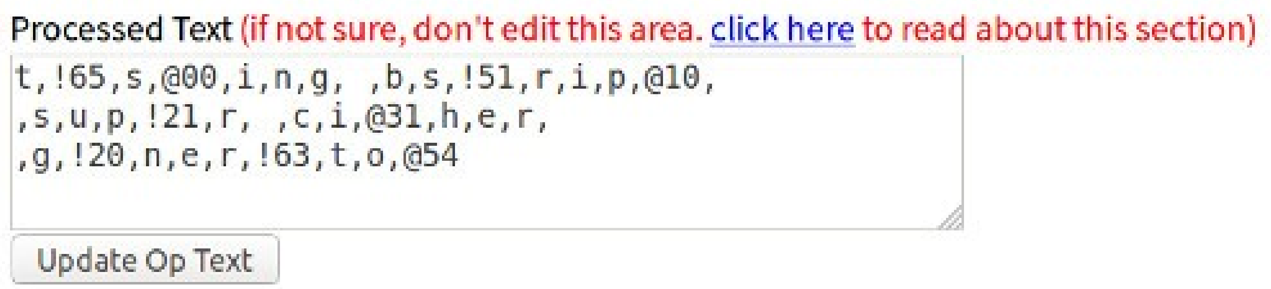
Symbol Map
Here you can modify the map of symbols. You can enter single characters or string of characters.

Key Code
Once you have designed a character map and operations maps you like, copy this code. You can paste it in here and press “update key code” to reload it later.

Operator 1 Map
A map for the operator which is drawn above the symbols. There are some preset operator maps that can be loaded.

Operator 2 Map
A map for the operator which is drawn above the symbols. There are some preset operator maps that can be loaded.

Processed Text
This is layer of intermediate array values for processing.



Processed Text (if not sure, don't edit this area. click here to read about this section)

```
t,!65,s,@00,i,n,g, ,b,s,!51,r,i,p,@10,
,s,u,p,!21,r, ,c,i,@31,h,e,r,
,g,!20,n,e,r,!63,t,o,@54
```

You can reference it to see how the text is processed and even modify it directly to manually draw symbols and operations.

The format is comma delimited (commas between symbols).

Operations are written as “!XY” X and Y are the positions of the input symbols in the word.

Ultra Secure Bscript

Finally we can create an ultra secure Bscript cipher by applying a base operation to everything. Some operations like XOR or Mod 4 Addition have a special feature, every input can be come any output. No matter what the first input is there is always a second input for any desired output.

Ultra secure Bscript uses one of these operations on every pair of letters. To decode it just apply the operation to each pair and take the new string of characters.

Te encode a word first chose any random character, then choose the appropriate character to create the desired character. If you chose a valid operation, such as XOR, then in “A OP B = C” no matter what A is there is always a B to produce any C.

There is an on-line web tool to help you encode a string at http://www.dscript.org/bscript/ultra.php.

You can also add the standard operation method on top of ultra secure Bscript. Just use “process word” tools after you have loaded it into the “processed text” field at the bottom of the genertor.

Decoding

To decode a string just apply the operation to each pair of letters.

XOR

Same = 0 (1 xor 1 =0 , 0 xor 0 = 0)

Same = LEFT / CORNER

LOOP xor LOOP = CORNER
CORNER xor CORNER = CORNER

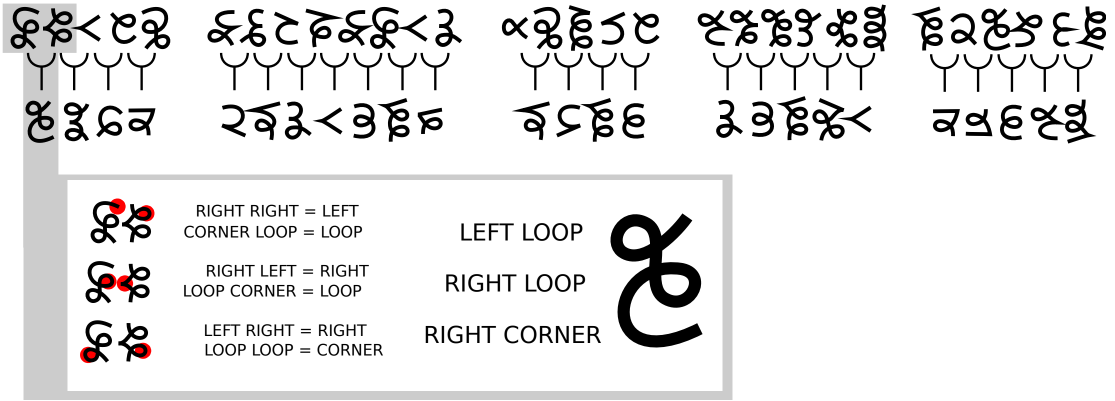
LEFT xor LEFT = LEFT
RIGHT xor RIGHT = LEFT

Not same = 1 (1 xor 1 =0 , 0 xor 0 = 0)

NOT Same = RIGHT / LOOP

LOOP xor CORNER = LOOP
CORNER xor LOOP = LOOP

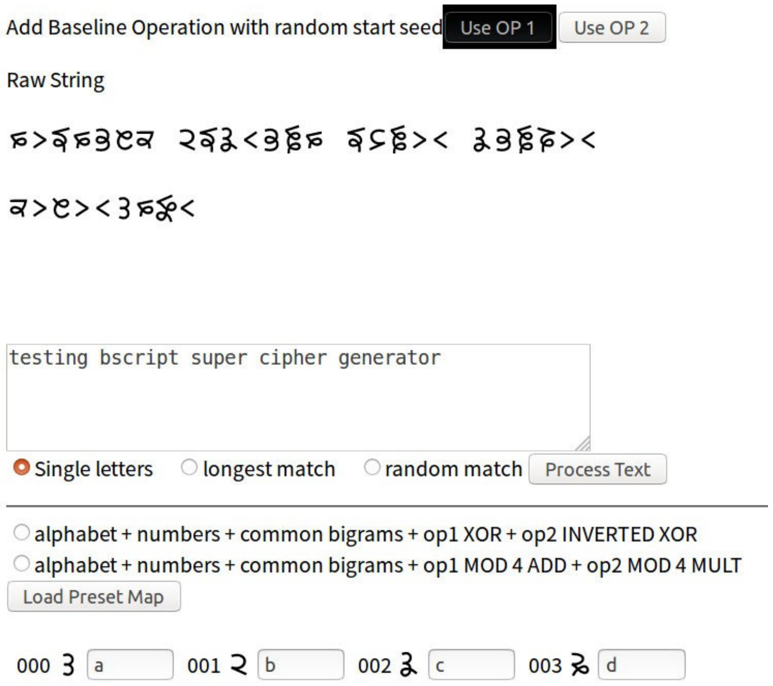
LEFT xor RIGHT = RIGHT
RIGHT xor LEFT = RIGHT



Encoding

To encode a string you can do it manually by starting each word with a random character, and then adding the appropriate characters to produce the word one pair at a time.

Or go to http://www.dscript.org/bscript/ultra.php
You can import your symbol map and operation map
Then type in your text and press process text, this will produce raw Bscript text
Next press on of the op buttons at the top, this will use that operation to encode the text



Add Baseline Operation with random start seed

Raw String

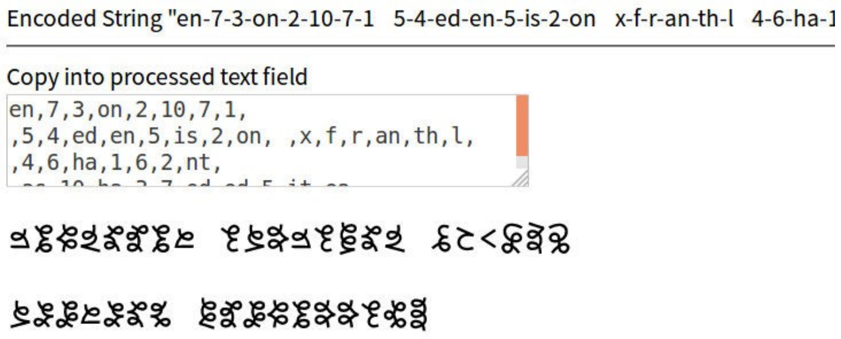
```
testing bscript super cipher generator
```

● Single letters ○ longest match ○ random match

○ alphabet + numbers + common bigrams + op1 XOR + op2 INVERTED XOR
○ alphabet + numbers + common bigrams + op1 MOD 4 ADD + op2 MOD 4 MULT

000 a	001 b	002 c	003 d

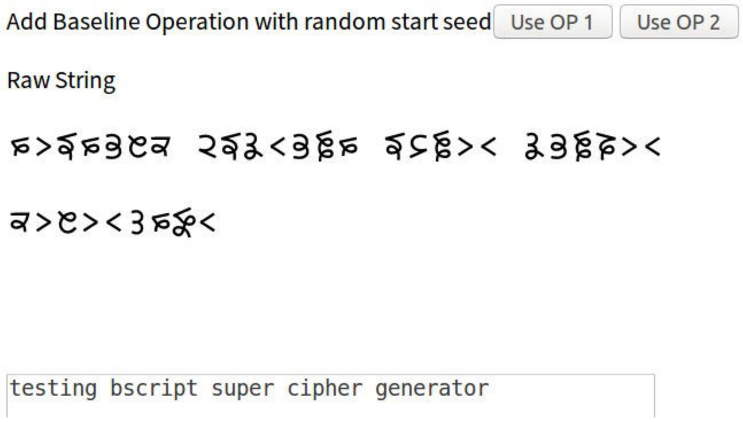
After clicking one of the OP buttons, that operation will be used to encode it



Encoded String "en-7-3-on-2-10-7-1   5-4-ed-en-5-is-2-on   x-f-r-an-th-l   4-6-ha-1

Copy into processed text field

```
en,7,3,on,2,10,7,1,
,5,4,ed,en,5,is,2,on, ,x,f,r,an,th,l,
,4,6,ha,1,6,2,nt,
```



Add Baseline Operation with random start seed

Raw String

```
testing bscript super cipher generator
```

On the top you will see what the encoded version would read if not decoded.

Next there is a text box with the encoded text. You can copy this and past it into the “processed text” box at the bottom of the main Bscript generator. You can then add operations on top.

Below that is the image of the encoded Bscript text.

Resources / Links

Bscript On-line tools : **http://dscript.org/bscript**
Source code for the on-line tools **http://dscript.org/bscript.zip**

More Technology-Art and Constructed Scripts at **http://www.dscript.org**

Free for all to use & modify, even free for commercial use

If you enjoy this script you may enjoy some of my other scripts.

Uscript The Universal Language : http://dscript.org/uscript.pdf

Uscript is a logographic language based on math and physics designed to be understandable by any intelligent life in the universe.

Dscript 2D writing system : http://dscript.org/dscript.pdf

Dscript is a 2D writing system, it allows words to be writen as strings and characters that are still legible. It is not ideal for computer use, and in fact is very resistant to OCR (computer Optical Character Recognition).

Cscript Computer-Human Bi-freindly Script : http://dscript.org/cscript.pdf

Cscript is designed to be easy to OCR, easy to read and write, and have several layers of compression to allow for spatially dense writing.

Escript Electronic script for Low-res pixel display Script : http://dscript.org/escript.pdf

Escript is optimized for both handwriting and low resolution pixel display.

Chemical Calligraphy Basics : http://dscript.org/chem.pdf
Chemical Calligraphy Advanced / Artistic : http://dscript.org/chem2.pdf

Chemical calligraphy is a derivative of Dscript. It allows chemical structures to be drawn as symbols. The basics pdf introduces the basic principles, and the advanced pdf demonstrates how to use it as an artistic and mnemonic device to help when learning chemical structures and organic chemistry.

WireScript : http://dscript.org/wirescript.pdf

WireScript allows text to be written with wires in 2D and 3D. WireScript turns text into physical 3D art.

NailScript : http://dscript.org/nailscript.pdf

NailScript is a way to write text with a hammer and nails. By hammering nails into wood or other materials very durable and long lasting text can be written.

Based on work by Matthew DeBlock at **http://dscript.org**
matthew@dscript.org