

Cryptography, Math and Programming with Cryptol

(and maybe some Python)

Dylan McNamee

Chapter 1

Getting started

This book is a guide to learning about cryptography, the math that cryptography is built on, and how to write programs that implement cryptographic algorithms. Don't worry if any of that sounds too complicated: it's all explained inside. Also don't worry if it sounds too boring: these topics are surprisingly deep and interesting – there's a lot of cool stuff even jaded students (of all ages!) can learn.

At the beginning of each chapter, we'll describe what you'll know by the end of the chapter. If you feel like you already know that stuff, try skimming the chapter (don't skip it!) to make sure that you do, and slow down and read the new stuff.

At the end of this chapter you'll know what resources you'll need to complete the activities in the book, and you'll have encrypted and decrypted a message using the Caesar cipher. Let's get going!

What you'll need

You'll need a sense of curiosity. You'll also need a dedication to actually doing the exercises: if you just read this material you'll only get a small fraction of the benefit of doing. You'll need access to a computer. It can be a Windows, MacOS or Linux computer - they're all fine.

Having a group of people to work with is a good idea. A lot of the activities require serious thought, and it's totally normal to get stuck (often). If you work with a group of people, when one of you gets stuck, the others can help out. Not by giving answers, but by nudging in productive directions.

Often all it takes to do this is to ask “What are you working on? What have you tried? Is there anything you haven’t tried yet?” Explaining the answers to these questions to another person is often enough to get unstuck.

Why cryptography?

Cryptography is the mathematics of secret messages. The popularity and pervasiveness of social media has caused some people to comment “nothing is secret any more.” But is that really true? People share photos taken in restaurants all the time, but is it a good idea to share a photo of the credit card you used to pay for your food? What about sharing your social media passwords? Needless to say, privacy and cryptography are both interesting, and are related to each other in subtle ways.

Let’s get started

Print out and assemble the Reverse Caesar Cipher kit that comes with the book. The Caesar Cipher is one of the earliest known ciphers, used by Caesar to communicate orders to distant generals. The idea is that if the messenger was intercepted by foes of Caesar, that they wouldn’t learn any secrets from the message they carried.

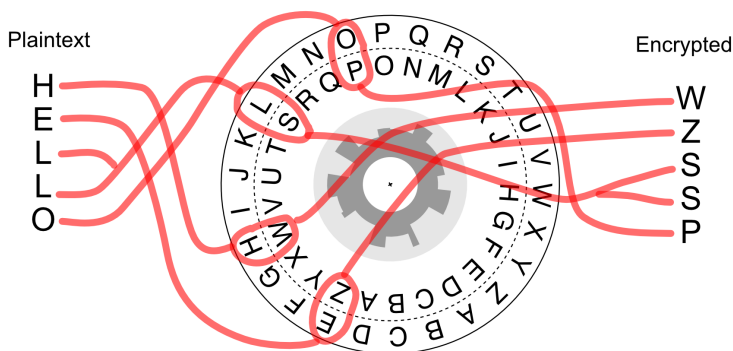


Figure 1.1: Encrypting “HELLO” with the Caesar Cipher. Key is $A \leftrightarrow D$

Follow the instructions on the kit to decode the following punchlines:

I wondered why the ball was getting bigger ... (Use the key: $J \leftrightarrow M$)

CORI NC ONC JR

What do you call a counterfeit noodle? (Key: F↔O)

TG LHETBAT

A backward poet ... (Key: H↔H)

SXGVKW GBTKXWK

If you managed to decrypt all three of these, congratulations - that's a lot of work! When I use a Caesar's Cipher wheel, it takes about 3 seconds per letter to encrypt or decrypt a message. At that rate, it would take about an hour to encrypt a whole page of text, which is way too long.

Given how tedious it is to decrypt, even when you know the key, it's not too hard of a stretch to imagine Caesar thinking this cipher was good enough. Now that we have computers, it's a lot easier to encrypt and decrypt messages, and the Caesar Cipher is not close to good enough. We'll learn about how "real world" cryptography works later in this book.

Things to ponder

1. you may have noticed that the cipher in this chapter has been called the Caesar Cipher but the cardboard version is called the Reverse Caesar Cipher. What do you think the difference between them is? Is there an advantage or disadvantage to either of them?
2. if it took you one minute to try to decrypt a message using a key you guessed, how long would it take, on average, to decrypt a Caesar Cipher message whose key you don't know¹? If it takes a computer 1 millisecond try one key on a message, how long, on average, would it take to decrypt a message without knowing the key?
3. how much more secure would it be to have weird symbols (Greek letters or Egyptian hieroglyphics), instead of letters, for the cypher-text in a Caesar's Cipher? Explain your answer.
4. *key distribution* is the challenge of getting the secret key to your friend. One way to distribute a key would be to include it in some hidden way in the message. Come up with a few ways you could do this with the Caesar Cipher. Another way would be to agree on a shared key when in the same room as your friend. What are some of the advantages and disadvantages of these two approaches?

¹Hint: first, how many different possible keys are there? It's safe to guess that on average, you'll have to try half of them before guessing the right one.

Take-aways

You've thought about why cryptography is important. You know how to encrypt and decrypt messages using the Caesar Cipher. You have thought about how secure it is, including the aspects of key distribution.

Chapter 2

Encoding data into bits

“There are 10 kinds of people in the world: those who know binary and those who don’t.”

Now that you’ve seen encryption and decryption at work, it’s time to learn how computers do it. Our Caesar’s Cipher wheel is a paper computer which has an alphabet of 26 elements. You’ve heard (most likely) that computers work with ones and zeros. One’s and zeros are not very helpful by themselves, so people figured out how to represent integers, floating point numbers and all of the letters in all of the languages around the world using only ones and zeros

The process of representing one set of things (integers, for example) using another set of things (sequences of ones and zeros) is called *encoding*. *Decoding* is reversing the process; getting back the original information from the new representation. In this chapter, we’ll learn how to encode and decode unsigned and signed integers, simple Latin alphabets, as well as the rest of the alphabets in the world.

Encoding integers

If the joke at the beginning of this chapter makes sense, and you know about *number bases*, encoding integers using ones and zeros is simply converting to base 2, and you can skip to the next section. To learn what this means, and why that joke isn’t leaving out eight kinds of people, read on¹.

¹ Note that we didn’t promise we’d convince you this joke is funny, only that you’ll understand what it’s getting at.

We're so used to seeing a number like 533 and understanding it to mean "five hundred and thirty-three" that we forget that it's an encoding of a numeric value into the symbols 0, 1, 2 ... 8, 9. Reading from right to left, the n^{th} digit is the 10^{n-1} -place². So deconstructing our example number we get:

Exponent	10^3	10^2	10^1	10^0
Value	1000	100	10	1
Digit	0	5	3	3
Digit value	0	500	30	3

So finally we get $0 + 500 + 30 + 3 = 533$.

Binary representations of numbers

Encoding numbers in binary is the same recipe, but with 2 as the base of the exponent instead of 10. Each place (digit) can either have a 1 or a zero in it. As a result, you need more digits to represent the same values, but it works out.

Exponent	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Value	512	256	128	64	32	16	8	4	2	1
Digit	1	0	0	0	0	1	0	1	0	1
Digit value	512	0	0	0	0	16	0	4	0	1

In this case we get $512 + 16 + 4 + 1 = 533$.

To convert a number into binary, take the largest digit that isn't bigger than your value, and set it to 1, then subtract that digit's value from your value and repeat down the line, not forgetting to put in 0's for the values you don't want to add.

Adding binary numbers is super-easy. It's a lot like the addition you're used to, with carrying and everything, except simpler. If both places are 0, the sum is 0. If either is 1, the sum is 1. If both are 1, the sum is 0, and you carry 1. When you include the carry, the rule is the same, except sometimes you have both 1 along with the carry, in which case the sum is 1 and the carry is 1.

Here's a four-bit addition of 2 and 3:

²Remember that $10^0 = 1$


```

    1   <- carry
  0010
+ 0011
-----
  0101

```

If you're talking about numbers in different bases, and it's not clear which one you're referring to, it's common to include the base in the subscript after the number. So the joke at the beginning of the chapter would be "There are 10_2 types of people..."³.

Working from the right, $0 + 1 = 1$ with no carry, then $1 + 1 = 0$ with carry, and finally we have a carry with $0 + 0$, so that's 1. 101_2 is 5, which is what we're hoping for.

Representing negative numbers:

This section is a deep-dive. You can skip it the first time through - but if you get bored or ever get curious, come back here for some cool stuff.

The encodings we've discussed so far are only for positive numbers. If you want to represent negative numbers, there are a few options, but one fairly universally agreed-on best way to go.

The obvious (but not best) way is to reserve the top-most bit to represent negation. If the bit is 0, the rest of the bits are a positive number, if it's 1, the rest of the bits are interpreted as a negative number. This encoding makes sense, but it makes arithmetic difficult. For example if you had 4-bit signed numbers, and wanted to add -1 and 3, you'd get

```

    11   <- carry
  1001
+ 0011
-----
  1100

```

This shows that if we apply our naive addition to $-1 + 3$, we get the unfortunate answer -4. Wouldn't it be cool if there were a way to store negative numbers in a way that the addition process we already know would just

³But that way kind of ruins the joke, doesn't it?

work out? It turns out that if you represent negative numbers by flipping the bits and adding one, you can do arithmetic using simple unsigned operations and have the answers work out right. This method of encoding signed numbers is called *two's complement*.

For example, to get a four-bit -1 in two's complement, here's the process:

```
Step 1: 0001  <- +1
Step 2: 1110  <- flipped
Step 3: 1111  <- add 1 is -1 in two's complement
```

Here's $-1 + 3$ again, in two's complement:

```
  111  <- carry
 1111 <- -1 (from above)
+ 0011 <- 3
-----
 0010
```

In the one's place, $1 + 1 = 0$ carry 1, then we have $1 + 1 + \text{carry} = 1$ carry 1, then we have $1 + \text{carry} = 0$ with carry, and the last digit is also $1 + \text{carry} = 0$ (and the carry goes away). You'll see the answer, $0010_2 = 2$, which is what we're hoping for.

Things to think about

1. What's the largest value you can represent with one base-ten digit? Two digits? n -digits?
2. What's the largest value you can represent with one binary digit? Eight digits? n -digits?
3. When we did $-1 + 3$, the carry bit got carried off the end of the addition. This is called overflow. In some cases (like this one), it's not a problem, but in other cases, it means that you get the wrong answer. Think about whether you can check whether overflow has occurred either before or after the addition has happened.
4. Two's complement is a slight change from *one's complement*, in which negative numbers just have their bits flipped, but you don't add a 1 afterwards. A big advantage of two's complement is that there are two ways to write 0 in one's complement: $10000\dots$ and $0000\dots$. Essentially you have a positive and a negative zero. Think about what problems this might cause.

5. What's the largest value you can represent in a two's complement 8-bit number? What's the smallest?

Why ones and zeros?

It's a reasonable question - *why do computers only use ones and zeros?* The oversimplified, but essentially correct answer is performance and simplicity. Making computers faster has been a goal since they were first invented. *Simplicity enables speed* is a common theme in computer engineering, and binary code is a great example. To represent values the voltage on a wire is either *high* (representing a 1) or *low* (representing 0). What exact voltage corresponds to high and low can vary. As systems get faster, the voltages that make a "1" tend to decrease. In current Intel CPUs, for example, it's common for a "1" to be as low as 1 volt. On older systems, it be as high as 13 volts.

Transistors are the building blocks that work with the voltages inside computers. They're essentially just switches that can be controlled by a voltage level. A transistor has an input, an output, and a controlling switch. It's easy to tell when a transistor is all the way "on" or "off", but measuring values in between is much more complex and error-prone, so modern computers don't bother with those, and instead just deal with "high" voltages and "low" voltages. Taking this approach has allowed us to create computers that can switch many *billions of times per second*.

Encoding text into ones and zeros

Now that you understand how numbers can be represented as ones and zeros, we can explain how text can be represented as sequences of numbers, and you can convert those numbers into bits.

It turns out that how to assign numbers to letters is pretty arbitrary. Until the early 1960's, there were a number of competing text → bits encoding systems. People realized early on that deciding on one system would let them communicate more easily between different machines. The most common text encoding, called ASCII, was agreed on in 1963, and was in wide use through the mid 1990's.

The table below show how ASCII represents the basic letters, numbers and punctuation. Each character is followed by its decimal ASCII code. There

are two “special” characters in the table, `sp` is the space character, and `del` is delete⁴.

sp	32	!	33	"	34	#	35	\$	36	%	37	&	38	'	39
(40)	41	*	42	+	43	,	44	-	45	.	46	/	47
0	48	1	49	2	50	3	51	4	52	5	53	6	54	7	55
8	56	9	57	:	58	;	59	<	60	=	61	>	62	?	63
@	64	A	65	B	66	C	67	D	68	E	69	F	70	G	71
H	72	I	73	J	74	K	75	L	76	M	77	N	78	O	79
P	80	Q	81	R	82	S	83	T	84	U	85	V	86	W	87
X	88	Y	89	Z	90	[91	\	92]	93	^	94	_	95
`	96	a	97	b	98	c	99	d	100	e	101	f	102	g	103
h	104	i	105	j	106	k	107	l	108	m	109	n	110	o	111
p	112	q	113	r	114	s	115	t	116	u	117	v	118	w	119
x	120	y	121	z	122	{	123		124	}	125	~	126	del	127

So the string “Hi there” in ASCII is: 72, 105, 32, 116, 104, 101, 114, 101.

Some exercises

1. Encode your name in ASCII.

ASCII has some clever design features. Here are some questions that may uncover some of that cleverness:

2. Is there an easy way to convert between upper and lower-case in ASCII? Think about the binary representations.
3. Is there an easy way to convert between a digit and its ASCII representation? Does the binary representation help here? What aspects of the ASCII encoding make this easy/difficult?

Encoding *all* languages: Unicode

This section is a deep-dive: you can do the rest of the book knowing only ASCII. On the other hand, if you like to know how things work under the hood, you'll enjoy learning how non-Latin web pages are encoded and transmitted.

⁴delete is more of an un-character, but it has an ASCII code

Up until the mid 1990's, computer systems that needed to process languages whose characters are not in the ASCII tables each used their own encodings. When the Internet and Word Wide Web started to gain adoption, people realized that they would have to standardize how these other languages encoded their alphabets into bits. The Unicode Consortium was the group founded to make those standards. They took the sensible approach of splitting the problem into two stages:

1. Enumerating all of the symbols that can be represented. This includes accents, special glyphs, and now also includes emoji. As of 2016, there are over 1.1 million different “code points” in the master Unicode table.
2. Devising efficient ways of representing sequences of those symbols as bits.

The hard work of the first stage is to come to agreement on which symbols go in (and which to leave out), what to call them, and how to organize them. The folks working on stage two have come up with a number of encodings, but the one that is most common on the Internet is UTF-8. The genius of UTF-8⁵ is that it's *backwards compatible* with ASCII. What that means is that if your text *does* fit in the ASCII table, the ASCII representation of it is also the UTF-8 representation of it. The key to making that work is that while ASCII is an 8-bit representation, the top-most bit of the ASCII table is always 0.

If you're decoding a UTF-8 stream of bytes, and you encounter any byte with its top bit off (i.e., its decimal value is ≤ 127), decode it as ASCII. If the top bit is on (the number is > 127), follow this procedure:

1. The first byte tells you how many bytes are in this character. Count the number of bits set before the first “0”-bit. That number is the number of bytes in this character. The remaining bits after the 0 are data. UTF-8 supports up to 4 bytes, so the longest (4-byte) UTF-8 character will start 11110 . . .
2. The remaining bytes are tagged with a leading “10” (so you can tell they aren't beginnings of characters), and the remaining 6-bits are data.
3. Concatenate the data bits into one binary number.

⁵UTF-8 was invented at Bell Labs by Ken Thompson, who co-invented Unix, and Rob Pike, who subsequently invented the Go programming language.

4. Look up that number in the Big Unicode Table.

Pretty cool!

An aside: Hexadecimal

Writing numbers in binary is tedious for mere humans⁶. It takes eight digits to count up to 128, after all! Writing them in decimal is convenient for us humans, but a downside is that there's no easy way to tell how many bits a decimal number has. Computer scientists have settled on *hexadecimal*, or base 16, to write numbers when the number of bits matters. How does one write a hexadecimal number? After all, we've only got ten digits, 0 -> 9, right? Well, as a convention we use the first six letters of the alphabet to represent the digits past 9. So counting to 16 in hexadecimal (or "hex" for short), looks like this:

1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10

Hex, just like decimal and binary, has a *one's place*, but the next bigger digit in hex is the *sixteen's place*⁷, so 10 in hex is 16 in decimal (also written as $10_{16} = 16_{10}$). A in hex is 10 in decimal. This means that one hex digit holds exactly four bits, and it takes two hex digits to hold a byte.

Finally, there are a number of ways of indicating what base a number is in. In addition to using the subscript of the base, like spoiling the joke with 10_2 , when you are writing numbers in ASCII and there's no way to write subscripts, instead we prefix binary numbers with 0b, and prefix hexadecimal numbers with 0x. If a number has no prefix or subscript, it's usually⁸ safe to assume the number is in base 10. Learning what hexadecimal looks like is important right now, because Unicode tables are all written in hex, as you're about to see.

Back to Unicode

Below is a table with three sample Unicode symbols. Each symbol has a long, boring unambiguous name, its graphical symbol (which can vary from font to font), its global numeric code in the master Unicode table, and finally how that number is encoded in UTF-8.

⁶computers, on the other hand, seem to thrive on tedium.

⁷and the next digit is the 256'ths place!

⁸when not telling nerdy jokes




Unicode name	Anticlockwise Gapped Circle Arrow	Bicycle	Pile of Poo
Symbol			
Numeric code	U+27F2	U+1F6B2	U+1F4A9
UTF-8	E2 9F B2	F0 9F 9A B2	F0 9F 92 A9

Figure 2.1: Some example Unicode glyphs, their official Unicode name, number and UTF-8 encodings

In the table above, the “U+” lets you know that the hex number that follows is the location in the Unicode table, and you see that the UTF-8 encoding is also written in hex. There’s a cool webpage at <http://unicode-table.com/en/> that has the whole table in one page. On the right of the page there is a live map with dots in the parts of the world where the characters visible on the current screen are used.

Let’s look at the UTF-8 for the Bicycle symbol: F0 9F 9A B2. In binary the F0 is 11110000. The four 1’s let us know that this UTF-8 code has four bytes total (this one and the next 3). The remaining 3 bytes are:

```

9      F      9      A      B      2
10011111 10011010 10110010

```

Remember that the beginning 10 in each byte lets us know these are the rest of this one symbol. If we take those off and concatenate the bits like this:

```

011111011010110010

```

Then breaking that up into 4-bit hunks (starting from the right), then converting each chunk into its Hex digit, we get:

```

Binary chunk: 01 1111 0110 1011 0010
Hex digit:    1  F    6    B    2

```

If you look at the Unicode Numeric code part of the table above, you’ll see that 0x1F6B2 is the code for Bicycle!

Independent study questions

If you're interested into learning more about how information can be digitally encoded, here are some questions you can research the answers to.

1. Two common ways of **encoding images** are pixel-based (or bitmap) and vector-based:
 - a. The main aspects of **pixel-based** encoding are resolution (how many pixels there are in the image), how to encode colors (the value at each pixel), and compression (e.g., to reduce the storage for simple scenes like a plain blue sky). Common pixel-based formats are PNG, JPEG, and GIF.
 - b. The main aspects of **vector graphics** are what *primitives* to provide, which are the shapes that are supported built-in (lines, curves, circles, rectangles) vs. which ones need to be assembled from sequences of primitives, what the *coordinate system* for describing shapes is, and what the *syntax* is. Vector graphics formats tend to more-resemble programming languages, and are often in human-readable ASCII. Common vector-based formats are PDF, SVG, and PostScript.

What's an image encoding method you know about? Use Google to find a specification for that format, and Write down how files in that format are structured. Most formats have a *header* which provides *metadata* about the file⁹.

2. **File archives** are encodings that combine a bunch of files and folders into one file that can be sent by email, or downloaded from a website, etc., and then *unpacked* at the other end. Archive formats often include the ability to compress the files as well. It's often surprising which file formats are archives. For example, most word processing document formats are file archives, to allow you to include graphics. Installers for most systems are also archives, such as Windows MSI files, MacOS DMG files, and Linux RPM files. Early archive formats include TAR and ZIP, which were invented more than 30 years ago, but are still used every day.

If you know a particular file archive format, look it up on the Internet and write it up in a page or so.

⁹The word metadata literally means "data about data", which particularly makes sense in this context

Take-aways

You've learned about how to encode data of different types (numbers, characters) into binary representations. You've learned some binary arithmetic, and why 10_2 is 2_{10} . Finally you've learned that nerds (the author included) can have a terrible sense of humor.

Chapter 3

Introducing Cryptol

Now that you understand how data can be represented in bits and have been introduced to cryptography using a paper computer, you're ready to learn a computer language that was designed for implementing cryptographic *algorithms*. An algorithm is a careful description of the steps for doing something. Algorithms themselves are math, but when you implement them, they're programs. The Caesar cipher from chapter one is a simple cryptographic algorithm. In this chapter, you'll learn about the programming language *Cryptol*, which was designed to make cryptographic implementations look as much like their mathematical algorithm description as possible. By the end of this chapter, you'll understand how to read and write simple cryptographic programs using Cryptol, and will be ready for the more complicated algorithms in the next chapter.

Installing and running Cryptol on your computer

How exactly to install and run Cryptol depends on whether your computer is running MacOS, Windows or Linux. Instructions for all three are provided on the Cryptol website, which is <http://cryptol.org>

You're ready to go with the rest of the chapter (and book) if you can run Cryptol, and follow along with the session below. In the examples below, **what you type will be in bold**, and what the computer types will be in non-bold (like this).

start cryptol, however you're supposed to on your system



Loading module Cryptol

```
Cryptol> 4 + 2
```

```
Assuming a = 3
```

```
0x6
```

```
Cryptol>
```

Here you've asked Cryptol to evaluate $4 + 2$, and it responded with `0x6`, which is a hexadecimal answer, but it's still just 6.

Types of data in Cryptol

One of Cryptol's main features is that it is very careful about how data is represented. How data is laid out is an example of a *type* in Cryptol. When you're talking about the type of something, often you'll see the thing, a colon (`:`) and then a description of its type

For example, the type of the hex constant `0xFAB` would be written:

```
0xFAB : [ 12 ]
```

You could read the above as “the type of the hex constant `FAB` is a sequence of twelve bits.” Cryptol can also talk about sequences of sequences. For example:

```
[ 0xA, 0xB, 0xC, 0xD, 0xE ] : [ 5 ][ 4 ]
```

I would call this “a sequence of five elements, each having four bits”. You can ask Cryptol what the type of something is with the `:t` command, like this:

```
Cryptol> :t 0xAB
```

```
0xab : [ 8 ]
```

Here we've said “Hey, Cryptol: what's the type of Hex `AB`?” and Cryptol replied (in a friendly robotic voice “Hex `AB` has the type *a sequence of length 8 of bits*”).

What do you think the type of a string of text should be? For example, what should the type of "hello cryptol" be? Assume the text is ASCII, not UTF-8. Stop reading for a minute and think about it.

Really, don't just read ahead, think about the type of the string "hello cryptol".

Okay, did you think about it? What did you come up with? One way to start answering questions like this one is outside in. By that I mean start by counting how many elements there are. In this case the length of "hello cryptol" is 13 characters. So, the start of the Cryptol type would be [13]. Next, think about the type of each character. Remember that ASCII characters are 8-bits each, so the rest of the type is [8]. To check your answer you can just ask Cryptol:

```
Cryptol> :t "hello cryptol"  
"hello cryptol" : [13][8]
```

Enumerations: sequence shortcuts

Cryptol has some fancy ways of creating sequences other than just having you type them in. One way is called *enumerations*. They're a short-hand way of writing sequences of numbers that increment in a predictable way. Here are some examples:

```
Cryptol> [1 .. 10]  
Assuming a = 4  
[0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9,  
0xa]
```

The *Assuming a = 4* is Cryptol helpfully telling you that it decided to use 4 bits per element of the sequence, because we weren't specific. From here on out, I'll leave out the *Assuming...* messages, unless they matter. Cryptol used 1 as the lower bound, 10 as the upper bound (which is 0xa in hex) and it incremented by one for each of the elements in between.

You can increment by a different amount by providing two starting elements. The step-value is the difference between them. For example:

```
Cryptol> [1, 3 .. 10] // the step here is 2 (because 3-1=2)  
[0x1, 0x3, 0x5, 0x7, 0x9]  
Cryptol> [10, 9 .. 1] // counting down (step = -1)  
[0xa, 0x9, 0x8, 0x7, 0x6, 0x5, 0x4, 0x3, 0x2,  
0x1]
```

Comprehensions: manipulating sequences

In addition to shortcuts for creating sequences, Cryptol has powerful ways of manipulating them, called *sequence comprehensions*. The way you write them in Cryptol is based on mathematical notation, so once you get used to them, you'll know some advanced math notation, too!

Here's how it works: a sequence comprehension is inside of square brackets, just like the sequences we've seen already. Then inside of that, there are two parts: first is a formula for building each element of the sequence. The formula is a mathematical expression that can have one or more *variables* in it. The second part is to define the values of the variables as being *extracted* from other sequences. This will make more sense with some examples:

```
Cryptol> [ 2 * x | x <- [1 .. 10]]
Assuming a = 4
[0x2, 0x4, 0x6, 0x8, 0xa, 0xc, 0xe, 0x0, 0x2,
0x4]
```

Reading the line we typed in goes like this: “Construct a sequence whose elements are two times x , where x is drawn from the list one to ten.”

Cryptol helpfully told us that it's decided the elements of the list are four bits each. Without being told otherwise, that's also the size of the elements of the new list, which is why our numbers wrap around to 0x0, 0x2, 0x4 at the end. If we want Cryptol to keep track of more bits in our output sequence, we can specify the type of the comprehension, like this:

```
Cryptol> [ 2 * x | x <- [1 .. 10]]:[10][8]
[0x02, 0x04, 0x06, 0x08, 0x0a, 0x0c, 0x0e, 0x10,
0x12, 0x14]
```

Here we've asked for the comprehension's type to be ten elements of eight bits each, and the result doesn't wrap around.

Defining functions

In math, *functions* describe a way of creating an *output* from one or more *inputs*. Functions in Cryptol are the same thing, and you can give them names if you want. Here's a picture example of a function named f , which takes two variables, x and y and returns their sum:

Inputs	Function	Output
--------	----------	--------

```

7 -----> x | f(x,y) = x + y | -----> 12
5 -----> y |

```

(TODO: make this pretty)

One way to define a function is with the `let` command, like this:

```

Cryptol> let double x = x + x
Cryptol> double 4
Assuming a = 3
0x0

```

What? Oh, yeah, Cryptol let us know it was working with 3 bits, because that's how many you need for 4, but $4+4$ is 8 which needs 4 bits, and the remainder is 0. The quickest way to get Cryptol to work with more bits is to use hex and add a leading 0:

```

Cryptol> double 0x04
0x08

```

Whew. That's better. Here's a definition of our function f , which takes two inputs:

```

Cryptol> let f x y = x + y
Cryptol> f 0x07 0x05
0x0c

```

If you're tired of reading hex, you can ask Cryptol to speak back to you in decimal:

```

Cryptol> :set base=10 ← use base 10 output
Cryptol> f 0x07 0x05
12

```

You can also call functions inside a sequence comprehension, like this:

```

Cryptol> [ double x | x <- [ 0 .. 10 ] ]
Assuming a = 4
[0x0, 0x2, 0x4, 0x6, 0x8, 0xa, 0xc, 0xe, 0x0,
0x2, 0x4]

```

And it should be no surprise that you can call functions from inside functions:

```

Cryptol> let quadruple x = double (double x)
Cryptol> quadruple 0x04
16 ← we still have output set to base 10

```

Functions on sequences

Now that you know about functions and sequences, it's time to learn about some functions that operate on sequences.

Extracting elements from sequences

The first one is called the *index operator*. That's a fancy way of saying getting the n^{th} element out of a sequence. It works like this:

```
1: Cryptol> let alphabet=['a' .. 'z']
2: Cryptol> alphabet @ 5
3: 102
4: Cryptol> :set ascii=on
5: Cryptol> alphabet @ 5
6: 'f'
```

On line 1, we created a variable called *alphabet*, which is a sequence of 8-bit integers that are the ASCII values of the letters of the alphabet. On line 2 we used the *index operator*, which is the @ symbol, to extract the element at location 5 of that sequence, which is 102. Since we wanted to see it as a character, on line 4 we used `:set ascii=on`, which tells Cryptol to print 8-bit numbers as characters. Finally, on line 5 we re-did the @ operation, which gave us `f`, which is the 6th letter of the alphabet. Why the 6th character and not the 5th? Cryptol, like most programming languages, uses *zero-indexing*, which means that `alphabet @ 0` is the first element of the sequence, `alphabet @ 1` is the second element and so on.

Cryptol also provides a *reverse index operator*, which counts backwards from the end of the sequence, like this:

```
Cryptol> alphabet!25
'a'
Cryptol> alphabet!0
'z'
```

What happens if you try to go off the end (or past the beginning) of a sequence? Let's try:

```
Cryptol> alphabet@26
invalid sequence index: 26
```

One more thing: @ and ! act a lot like functions, but they're called *infix operators*. The only difference between a function and an operator is that when you call a function, its name comes first followed by the values you

want the function to operate on (we call those its *arguments*). Operators only work with 2-arguments, and the operator name comes *between* the two arguments. All of the normal math operators you're familiar with are infix operators, like: $5 + 2 - 3$.

Reversing a sequence

Cryptol provides a function called `reverse`. Let's try it:

```
Cryptol> reverse ['a' .. 'z']  
"zyxwvutsrqponmlkjihgfedcba"
```

Pretty handy!

Concatenating sequences

The `#` operator combines two sequences into one sequence, like this:

```
Cryptol> ['a' .. 'z'] # ['A' .. 'Z']  
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

“Rotating” elements of a sequence

The `>>>` and `<<<` operators *rotate* the elements of a sequence n places. For example,

`['a' .. 'z'] >>> 1` returns `"abcdefghijklmnopqrstuvwxy"`. All of the elements get shifted 1 place to the right, but the ones that fall off the end *rotate* back to the beginning.

`['a' .. 'z'] <<< 2` returns `"cdefghijklmnopqrstuvwxyzab"`. Everything moves to the left two places, but the first two, which fall off the front, rotate around to the end.

Functions have types, too

This section is a deep-dive into Cryptol's fancy type system. You don't need to know this to complete the first few exercises, but it's really neat, and will help you understand some of the things Cryptol says to you.

We mentioned earlier in this chapter that Cryptol is very careful about the types of things. In addition to data, functions in Cryptol have a type. The type tells you how many arguments a function takes as input, and what type each of those arguments needs to have, as well as the type of the output. Just like for data, you can ask Cryptol what the type of a function is by using `:t`, like this:

```
Cryptol> :t double
double : {a} (Arith a) => a -> a
```

The way you read a function-type in Cryptol has two parts, which are separated by a “fat arrow” (`=>`). Before the fat arrow is a description of the types, and after the fat arrow is the description of the inputs and the output. Each of them is separated by a “normal arrow” (`->`). The last one is always the output. The ones before that are the parameters.

Looking at our type of `double`, we see that it operates on things that you can perform arithmetic on (`Arith a`), it takes one argument and produces output of the same type.

You can ask Cryptol about the types of an *infix operator* by surrounding it with parentheses, like this:

```
Cryptol> :t (+)
(+) : {a} (Arith a) => a -> a -> a
```

This says that plus takes two inputs, and produces one output, all of which are *Arithmetic*.

What’s an example of an input type that isn’t Arithmetic? Concatenation is one. Check this out:

```
Cryptol> :t (#)
(#) : {front, back, a} (fin front) =>
[front]a -> [back]a
-> [front + back]a
```

This is a bit complex: What it says is that `front` and `back` are both sequence-lengths, and that `front` is of finite length (`fin`). After the `=>`, it lets us know that the first argument has `front` elements, the second argument has `back` elements, and the output has `front + back` elements. The `a` everywhere lets us know that the sequence could be of anything: a single Bit, or another sequence, or whatever. They do all have to be the same thing, though.

Implementing the Caesar Cipher in Cryptol

Using what you've learned so far, let's implement the Caesar Cipher in Cryptol. Let's start by breaking down the process of encrypting and decrypting data using the Caesar Cipher.

Let's guess what the function declaration should look like. We know that the encrypt operation takes a key and a message, so the function declaration probably looks something like:

```
caesarEncrypt key message =
```

Let's talk about how we can represent the key. In Chapter 1, we talked about the key being something like $K \leftrightarrow D$, but that's hard to represent mathematically. If we straighten out our Caesar Cipher wheels into a line, it looks something like this:

```
zyxwvutsrqponmlkjihgfedcba <- outer wheel  
abcdefghijklmnopqrstuvwxyz <- inner wheel
```

If we then think about the *rotate* operator (\gg), we see that they do something really useful. For example, let's rotate the outer wheel by 3:

```
zyxwvutsrqponmlkjihgfedcba <- outer wheel  
xyzabcdefghijklmnopqrstuvw <- inner wheel: ['z' .. 'a'] >>> 3
```

Hey - that makes sense: and even the description (rotating the inner wheel by 3 positions) *sounds* like what we did with the paper Caesar Cipher.