
Cryptography, Math and Programming

Release 1

Dylan McNamee

Nov 28, 2016

CONTENTS

1	Getting started	3
1.1	What you'll need	3
1.2	Why cryptography?	4
1.3	Let's get started	4
1.4	Things to ponder	5
1.5	Take-aways	6
2	Encoding data into bits	7
2.1	Encoding integers	7
2.2	Why ones and zeros?	11
2.3	Encoding text into ones and zeros	12
2.4	Independent study questions	17
2.5	Take-aways	18
3	Introducing Cryptol	19
3.1	Installing and running Cryptol on your computer . . .	19
3.2	Types of data in Cryptol	20
3.3	Defining functions	23
3.4	Functions on sequences	25
3.5	Implementing the Caesar cipher in Cryptol	29
3.6	What we covered this chapter	34
4	Implementing More Complex Programs	37
4.1	Storing Cryptol programs in files	37

4.2	Creating a text file in a project directory	37
4.3	Implementing our next cipher: Vigenère	40
4.4	What we covered this chapter	45
Index		47

Contents:

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!

1.1 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.

1.2 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.

1.3 Let's get started

Print out and assemble the "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.

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: $F \leftrightarrow O$)

TG LHETBAT

Plaintext

Encrypted

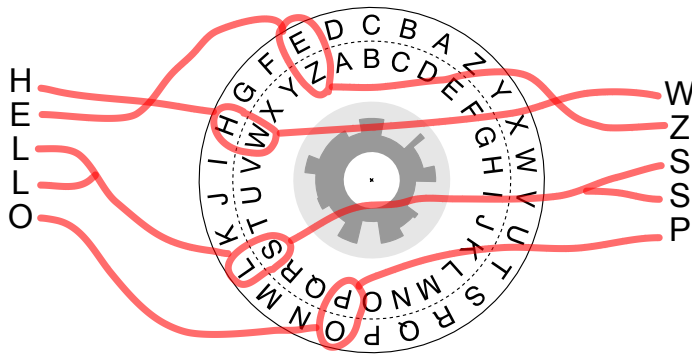


Fig. 1.1: Encrypting “HELLO” with the Caesar cipher. Key is $A \leftrightarrow D$

A backward poet ... (Key: $H \leftrightarrow H$)

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.

1.4 Things to ponder

1. you may have seen a Caesar cipher whose inner wheel and outer wheel go in the same direction, but the one in this book has

them going in opposite directions. What attributes do each version have? What advantages or disadvantages can you think between the two versions?

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 Egyption heiroglyphics), instead of letters, for the cyphertext 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?

1.5 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.

¹ Hint: 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.

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.

2.1 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.

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 $0 + 500 + 30 + 3 = 533$.

2.1.1 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.

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.

² Remember that $10^0 = 1$, and in fact anything to the 0th power is 1

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:

```
      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.

2.1.2 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

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

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 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.

2.1.3 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... and 0000.... 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?

2.2 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*.

2.3 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” charaters 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

⁴ delete is more of an un-character, but it has an ASCII code. So does “ring a bell” (which is ASCII 7). Kinda weird, isn't it?

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.

2.3.1 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?

2.3.2 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.

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

Pretty cool!

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

2.3.3 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 usually safe⁸ 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.

2.3.4 Back to Unicode

Below is a table with three sample Unicode symbols. Each symbol has a long, boring unambiguous name, its graphical symbol (which

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

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

⁸ except when telling nerdy jokes

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.




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

Fig. 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!

2.4 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**, or **bitmap** 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 *metatada* about the file⁹.

⁹ The word *metadata* literally means “data about data”, which particularly makes

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.

2.5 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.

sense in this context.

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.

3.1 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>


```
[ 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? 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]
```

3.2.1 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]``
```

3.2.2 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 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.

3.3 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 *parameters*, x and y and returns their sum:

Inputs	Function	Output
7 -----> x	$f(x, y) = x + y$	
5 -----> y		-----> 12

```
(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? $4 + 4 = 0$? 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 has two parameters:

```
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
```

3.4 Functions on sequences

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

3.4.1 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-based 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 two arguments, and the operator name comes *between* the two arguments. All of the normal math operators you're familiar with are infix operators, like: $5 + 2 - 3$.

***Arguments vs. parameters*:** when we talk about defining and calling functions, we've talked about both *arguments* and *parameters*, so you may wonder "what's the difference?" The answer is that *parameters are in a function's definition*, and *arguments are what you pass to a function when you call it*. So:

```
let foo x y = x - y    // x and y are the_
↳parameters of *f*
f 5 3                 // here we've passed 5 and 3_
↳as arguments to f
```

3.4.2 Reversing a sequence

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

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

Pretty handy!

3.4.3 Concatenating sequences

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

```
Cryptol> ['a' .. 'z'] # ['0' .. '9']  
"abcdefghijklmnopqrstuvwxyz0123456789"
```

3.4.4 “Rotating” elements of a sequence

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

`['a' .. 'z'] >>> 1` returns `"zabcdefghijklmnopqrstuvwxyz"`. 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 `"cdefghijklmnopqrstuvwxyza"`. Everything moves to the left two places, but the first two, which fall off the front, rotate around to the end.

3.4.5 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 is 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.

3.5 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:

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

To use the code wheel in this arrangement, look up a character from the top line, and the character directly below it is the encoded / decoded translation of that character.

If we think about the *rotate* operator (\gg), we see that it does something really useful. For example, let’s rotate the inner wheel by 4:

```
abcdefghijklmnopqrstuvwxyz <- outer wheel
dcbazyxwvutsrqponmlkjihgfe <- inner wheel >>> 4
```

This corresponds to the $A \leftrightarrow D$ key in the HELLO example in Chapter 1. It even makes sense: the description (rotating the inner wheel by 4 positions) *sounds* like what we did with the paper Caesar cipher.

At this point we’d *like to use* the index operator ($@$) to get the cyphertext from the inner wheel that corresponds to the plaintext on the outer wheel. The indexing operator needs to be a number, not a letter. For the index operator to do what we want, plaintext ‘a’ should be ‘0’, ‘b’ should be ‘1’, all the way up to ‘z’ should be 25. Let’s pause to think

about how to achieve that in Cryptol. First, remember that a character in Cryptol is already a number: its ASCII code. So, what if we subtract the ASCII code for 'a' from our plaintext character?

In ASCII, 'a' is 0x61, so 'a' - 'a' is 0, which is a good start. 'b' is 0x62, so 'b' - 'a' is 1, which is also what we're after. Finally, 'z' - 'a' is 25, so for that range of characters, it's good! Here's a simple function that takes an ASCII character and returns its index in the alphabet:

```
Cryptol> let asciiToIndex c = c - 'a'
```

Using this function to encrypt one letter would look like this:¹

```
Cryptol> let encryptChar wheel c = \  
wheel @ (asciiToIndex c)  
Cryptol> let codeWheel key = \  
reverse alphabet >>> key  
Cryptol> encryptChar (codeWheel 4) 'h'  
'w'
```

The `encryptChar` function takes a shifted wheel and a character `c`. It uses the index operator to extract the element from the wheel corresponding to the index value of the character. On the next line we defined `codeWheel` to be the reversed-alphabet shifted by our key. Finally we called our function. The first argument is our `codeWheel` with 4 as the key, and the second argument is our plaintext `h`. The output is `w` as we hoped.

Now we're ready to have Cryptol do this for every character in a string. Remember our sequence comprehensions? Here's how that comes together:

```
Cryptol> let encrypt key message = \  
[ encryptChar (codeWheel key) c | c <- message ]  
Cryptol> encrypt 4 "hello"  
"wzssp"
```

¹ Some of the examples on this page have backslashes (\) in them: it's because they're on more than one line: if you type the \, Cryptol will let you continue typing on the next line. Alternatively you can type it all on one line (and skip typing the \).

Hooray!

Now, what about decryption?

If you recall from Chapter 1, encryption and decryption are the same process. Let's test if that works:

```
Cryptol> encrypt 4 "wzssp"  
"hello"
```

Since that's not a satisfying name for a decryption routine, we can define `decrypt` in terms of our `encrypt` function:

```
Cryptol> let decrypt k m = encrypt k m  
Cryptol> decrypt 4 "wzssp"  
"hello"
```

Ah, much better. One thing to note here: in our definition of `encrypt`, the parameters were called `key` and `message`, but here we called them `k` and `m`. The reason that's not a problem is that when you're defining a function, you are free to name the parameters whatever you want - the only thing you have to remember is to use those same names in the body of your function.

This has been a huge chapter. If anything didn't make sense, go back and read it again, or ask a partner for help. We shouldn't go much further without really understanding what we've done so far. If Cryptol gives you mysterious errors instead of the output you expect, check what you've typed very carefully - we'll learn more about the errors Cryptol prints, and what you can learn from them.

3.5.1 Handling unexpected inputs

Let's try encrypting something new:

```
Cryptol> encrypt 7 "I LOVE PUZZLES"  
  
[warning] at <interactive>:1:1--1:30:  
  Defaulting type parameter 'bits'
```

```
                of literal or demoted expression
                        at <interactive>:1:8--
↪1:9
  to 3
  Assuming a = 7

invalid sequence index: 232
```

Egads - what just happened? When I see something like this happen, I first read the error message, then I think about what I did that could cause it. Starting at the top, the [warning] . . . tells you advisory things, not errors. That warning goes on for four lines, ending in `to 3`. The line after that is the normal helpful Cryptol telling you it's decided to use 7 bits for your ASCII string.

The problem is in that last line `invalid sequence index: 232`. We've tried to use the index operator (`@`) with an invalid argument. 232 is way bigger than 25 - where did that come from? We subtracted 'a' to make sure our indexes were all between 0 and 25, right?

At this point, it's time to start thinking about what we did wrong to cause this. Comparing this message to the one that worked, "hello", there are two main differences: our new message is in ALL CAPS, and it also has spaces in it. It turns out those are both problems we need to fix.

Let's start by handling upper case input. There are (at least) two ways we could do it. One is to have upper case input produce upper case output, and the other is to just make everything lower case. I think the second option is simpler, so let's do that.

Recall from Chapter 2's discussion about ASCII's clever design, that there's a simple way to convert between upper and lower case. Here are the Hex values of the ASCII codes for a, A, z and Z

+-----+	+-----+	+-----+	+-----+	+-----+
Character	A	Z	a	z
+-----+	+-----+	+-----+	+-----+	+-----+
Hex ASCII	0x41	0x5a	0x61	0x7a

```
+-----+-----+-----+-----+-----+
```

Hey, the difference between the upper and lower case values is exactly 0x20! If we want everything in lower case (WHO LIKES SHOUTING, REALLY?), if a character is lower than 0x61, we can add 0x20 to make it upper case. We use *conditional statements* to do that in Cryptol:

```
Cryptol> let toLower c = if c < 'a' then c + 0x20
↪else c
Cryptol> toLower 'I'
'i'
```

and just to make sure we didn't break already lower case input:

```
Cryptol> toLower 'i'
'i'
```

As you can see, a conditional statement has three parts: the *condition*, the *if-expression* and the *else-expression*.

Now we can use `toLower` to improve `asciiToIndex`:

```
Cryptol> let asciiToIndex c = (toLower c) - 'a'
```

And now we can encrypt text with upper and lower case (but without spaces):

```
Cryptol> encrypt 7 "iLOVEpuzzles"
"yvslcrmhhvco"
Cryptol> decrypt 7 "yvslcrmhhvco"
"ilovepuzzles"
```

Now, how to handle spaces. The usual way to handle spaces with the Caesar cipher (not in cryptography in general) is to pass them through. Sure, it makes the code weaker (you can see the length of words), but this part of the lesson isn't about good codes. To pass spaces through from the input to the output, the best place to do that is with a conditional in the `encryptChar` function:

```
Cryptol> let encryptChar wheel c = \
if c == ' ' then c else wheel @ (asciiToIndex c)
```

Let's test it, first on a space (since that's our new feature), then on an uppercase letter, and then on a lowercase letter:

```
Cryptol> encryptChar (codeWheel 7) ' '
' '
Cryptol> encryptChar (codeWheel 7) 'I'
'y'
Cryptol> encryptChar (codeWheel 7) 'i'
'y'
```

Yay, it looks like it'll work. Now let's encrypt and decrypt our original message:

```
Cryptol> encrypt 7 "I LOVE PUZZLES"
"y vslc rmhhvco"
Cryptol> decrypt 7 "y vslc rmhhvco"
"i love puzzles"
```

Wow - it all worked! If it didn't, go through the error messages, and see if you can figure out what happened.

3.6 What we covered this chapter

We covered a lot of ground this chapter:

- Launching Cryptol and asking about *types* of data with the `:t` command,
- *enumerations* are shortcuts for creating sequences, like `[1 .. 10]`,
- *comprehensions* are ways of manipulating elements of sequences,
- *functions* define how to create an output value from one or more inputs (called *arguments*),

- a number of functions that operate on sequences, like *indexing*, *reversing*, *concatenating*,
- finally, we implemented the Caesar cipher in Cryptol, step by step:
 1. converting ASCII characters to indexes,
 2. rotating the alphabet to make an encryption sequence,
 3. indexing the encryption sequence to encrypt one character,
 4. using a *comprehension* to encrypt a whole string,
 5. using *conditional expressions* to convert uppercase to lowercase,
 6. and handling the space character, ' ', by passing it through.

That's a lot of stuff - congratulations!

IMPLEMENTING MORE COMPLEX PROGRAMS

At this point, you’ve written some Cryptol at the command-line, but if you had to leave your machine, reboot, or whatever, you had to start from scratch. “That’s no good”, I hear you say, “there must be another way!” Indeed there is.

4.1 Storing Cryptol programs in files

In addition to typing commands at the Cryptol interpreter, Cryptol can load files that have programs in them. Programs are slightly different from the commands you’ve been typing so far. The main difference is that you don’t need `let` when you’re defining a variable (like `alphabet`) or a function (like `encrypt`).

4.2 Creating a text file in a project directory

The next thing you’ll want to do is create a directory (or folder) to keep your project files for this book. You’ll need to figure out how to edit *text files* on your computer. Text files

are different from, say, word processing documents, in that they do not have any formatting (no **bold** or *italics*, for example). They're literally just sequences of ASCII characters (or UTF-8 if you have a fancy editor). The command-line programs `vim` and `emacs` are still popular, even though they look like Wargames-era technology¹. More modern text editors include *Sublime Text* (<https://www.sublimetext.com/>), *nano* (<https://www.nano-editor.org/>), and many others. If you're on Windows, and are in a pinch, both *Notepad* and *Wordpad* can "Save As" *plain text*.

4.2.1 Exercise: rewriting our Caesar cipher program

Start this exercise by creating your project directory. On a Unix systems (like Linux or MacOS), you'll want to start up a Terminal window, on Windows start a shell window (type `cmd` in the search bar), and type something like this:

```
$ mkdir CryptoBook
$ cd CryptoBook
```

Then, using whichever editor you choose, create a file called `caesar.cry` inside that directory. It should contain the following:

```
// Caesar cipher
alphabet = ['a' .. 'z']
toLower c = if c < 0x61 then c + 0x20 else c
asciiToIndex c = (toLower c) - 'a'
encryptChar wheel c = if c == ' '
    then c
    else wheel @ (asciiToIndex c)
codeWheel key = reverse alphabet >>> key
encrypt key message =
```

¹ no coincidence, either: the original versions of these programs were written almost 10 years before Wargames came out. They're still popular because they're very powerful, but that's a lesson for another book.

```
[ encryptChar (codeWheel key) c | c <- message_
↪]
decrypt key message = encrypt key message
```

that's proving difficult, ask a partner or Google for *command line navigating files* for the operating system you're running on.

Let's see our encrypted string:

```
Main> :set ascii=on
Main> encrypt 4 "using files now"
"jlvqx yvszl qph"
```

Hooray! You'll never have to type the Caesar cipher again.

4.2.3 Exercise: motivating stronger encryption

Now that you can have a program perform the Caesar cipher for you, it's a simple thing to write a program that cracks a message that has been encoded with the Caesar cipher: just try all possible keys, and see which one decodes into an intelligible message.

But before using brute force, let's look at the following encoded message, and see if there are any clues to decryption:

```
"seh zldy wuxkahz pdse seh jlhtlu jdwehu dt sehuh
luh xyan sphysn tdo wxttkakah bhnt"
```

Before reading further, come up with two approaches you'd take to decrypting this message, and try them out.

Now that you've done that approach, use brute force to crack this message:

```
"wlszknzlosgey fzyeaylknzqlsfmoaosqlhozzob"
```

4.3 Implementing our next cipher: Vigenère

Given the previous exercise (please actually do it – it's a lot of fun, and very informative), it's probably occurred to you that the Caesar cipher

leaves a lot of room for improvement. You may have even thought of some changes to the Caesar cipher that would make it harder to crack, given the tools you've already developed.

Since we have brute force as an option, the main way to combat that attack is to greatly increase the number of guesses a brute force attacker will have to try. A simple approach to doing this is to have the key be a sequence of shift amounts. This is essentially the idea behind the Vigenère cipher. It was so successful at thwarting decryption, it was used for almost 300 years - from the 1500's through the 1800's, and was known for a lot of that time as "the indecipherable cipher".

Here's how it works:

Take a plaintext message, like "how do you like my fancy new cipher" and a key, like "thisismyfancykey", and to encrypt the i^{th} character, use the Caesar cipher with the i^{th} character of the key to specify the shift amount. To translate an ASCII key into a shift amount, we do the classic "subtract the ASCII value of a from the ASCII value of the key character".

The last detail is what to do if the message is longer than the key. What the Vigenère cipher does in this case is to "wrap around" to the first character of the key, and so on. In mathematical terms, this is known as *modulo arithmetic*. You're already familiar with modulo arithmetic from how we read clocks: there are 24 hours in a day, but our clocks only go to 12. For the hour past noon (or midnight) we "wrap around" to 1, and the next hour is 2, and so on.

Cryptol offers us a couple ways of expressing this notion of wrapping around the key. The first one is to use modulo arithmetic on the index. The second one is to create an *infinite sequence*, which consists of the key appended to itself over and over. Using this infinite sequence, we never have to worry about running out of key. This latter technique is a simple version of what we call *key expansion* in more sophisticated ciphers. Here's one way to implement the Vigenère key expansion in Cryptol:

```
expandKey key = key # expandKey key
```

In that one line of code, there are a number of things to explain! First,

it seems a bit magic (or cheating) that we're using `expandKey` in the definition of `expandKey`. This trick is a technique in programming called *recursion*. (POINT AT A RECURSION SECTION / RE-SOURCE)

The second thing we need to explain is “how / why this doesn't run forever, as soon as you expand any key - we haven't told Cryptol ever to stop!” That's right, we haven't, but let's give it a try anyway. First, copy your `caesar.cry` into a new file called `viginere.cry` (because we'd like to reuse a lot of the code in `caesar.cry`, and I promised you wouldn't have to type it in again). Second, add the above definition of `expandKey` to the end of your `viginere.cry`. Finally, start up Cryptol:

```
$ cryptol viginere.cry

      _ _ _ _ _ _ _ _ _ _ | _ _ _ _ |
 / _ | ' _ | | | | ' _ \ | _ / _ \ |
 | ( _ | | | | _ | | ) | | | ( ) | |
 \ _ _ | _ | \ _ , | . _ / \ _ \ _ / | _ |
           | _ _ / | _ | version 2.4.0

Loading module Cryptol
Loading module Main

Main> :set ascii=on
Main> let myXkey = expandKey "HELLO"
Main> myXkey
['H', 'E', 'L', 'L', 'O', ...]
Main> myXkey@1000
'H'
Main> myXkey@1001
'E'
```

Let's go through that line-by-line. First, we `:set ascii=on` so we can see the ASCII key strings. Second, we defined a temporary variable `myXkey` to be the result of expanding "HELLO". Next we asked Cryptol to show it to us. But rather uninterestingly, it just showed us the first five characters, but at the end it shows "...", signifying the list goes on. So, we decide to test Cryptol's expansion by indexing our

myXkey at index 1000, which happens to be an 'H', and then the next one, 1001, which is the expected 'E'. So far, so good!

The last Cryptol feature you'll need to learn in order to implement the Vigenère cipher is how to access two sequences at once in a sequence comprehension. We need this because we need to access both the next character of the expanded key stream and of the message in order to produce the next character of the ciphertext. Cryptol's way of doing this is called a *parallel comprehension*, and it looks like this:²

```
Main> [ encryptChar (codeWheel k) c \
      | c <- "hi there" \
      | k <- expandKey [0 .. 10] ]
"ss jwaoc"
```

One way to think of how parallel comprehensions work is like a zipper. When you zip up your jacket, the pull joins the elements (teeth) from each side and combines them (zips the teeth together). The length of the resulting sequence is the shorter of the two lengths of the sides of the zipper. In this case, it's the length of our message, "hi there", because our expanded key has infinite length.

```
h   i       t   h   e   r   e
0   1   2   3   4   5   6   7   8   9   10   0   1
→ ...
-> zip along the elements (TODO: make this pretty)
```

This also explains how Cryptol doesn't have to run forever when you ask it to define an infinitely expanded key: it only evaluates elements of a list as you ask for them. As long as you ask for only a finite number of them, Cryptol only evaluates that many of them. This way of approaching infinite sequences and "evaluate on-demand" is called *lazy evaluation*; it's a really powerful feature of Cryptol, and you'll see it's used quite a bit.

This is *almost* the Vigenère cipher: we're shifting our plaintext a different amount each time, but we're getting the shift amount from [0

² Note the \'-s - you can either type this all on one line, or use \'-s and include newlines where they appear here.

... 10] repeated, instead of a key string turned into indexes.

See if you can finish the implementation of the Vigenère cipher based on what you know about Cryptol now.

It should start like this:

```
viginere key message =  
  ... you fill in the rest
```

4.3.1 Exercises

1. Use your implementation of the Vigenère cipher to encode and decode some messages. Notice some of the improvements using longer keys makes.
2. Decrypt the following message using the key: "thisphraseismykey"

"qrucuvso dtoezje yzspd yordwt llsarvnij
jzrwyam"
3. One kind of attack against a code is when you know what some or all of a message is, and use that knowledge to learn something about the key. This was used during World War II when the Allied cryptanalysts guessed the word "weather" would appear in a German message that was encrypted with the Enigma machine. This technique is called a *known plaintext attack*.

Think about how you could learn the key used in a Vigenère-encrypted message if you knew that a message started with the plaintext "At the tone the time will be...". started with the following ciphertext: "gg njc fyjg doa joec jyfv xi".

4.4 What we covered this chapter

We started by learning how to program in Cryptol using files, and how to run Cryptol using a file you wrote. Next, we discussed some of the weaknesses of the Caesar cipher, and even did some codebreaking that uses those weaknesses. This lead to a discussion of increasing key length, which is exactly what the Vigenère cipher does. We learned about *key expansion*, and did it in Cryptol using *recursion*. We combined the expanded key with the message using *parallel comprehensions*, and learned that Cryptol uses *lazy evaluation* to avoid infinite loops when sequences are infinitely long. Finally, you used the techniques learned in this chapter to implement your own Vigenère cipher. The exercises gave you a chance to exercise your new code, and learn about the *known plaintext* technique to learn a key from an encoded message.

A

- algorithm, 19
- archives, 17
- arguments, 26
- arguments vs. parameters, 26
- ASCII, 12

B

- binary, 8
- binary addition, 9
- bitmaps, 17

C

- conditional statements, 33
- Cryptol, 19

D

- decimal, 8
- decoding, 7
- defining functions, 23
- dmg file, 17

E

- encoding, 7
- enumeration, 21

F

- file archives, 17
- function, 23
- function definition, 23

H

- hexadecimal, 15

I

- index operator, 25
- infinite sequences, 41
- infix operators, 26

K

- key distribution, 6
- key expansion, 41

L

- lazy evaluation, 43

M

- metadata, 17
- modulo arithmetic, 41

N

- number bases, 7

O

one's complement, 11
overflow, 11

P

parallel comprehension, 43
parameters, 23, 26
performance, 11
pixels, 17

R

recursion, 41
reverse index operator, 26
rotate operator (>>>), 29
rpm file, 17

S

sequence, 21
sequence comprehension, 22

T

tar file, 17
text editors, 37
text file, 37
transistors, 11
two's complement, 10
type, 20

U

Unicode, 13
UTF-8, 13

V

variable, 22
vector graphics, 17
Vigenère cipher, 40

Z

zero-based indexing, 25
zip file, 17