

---

# **Cryptography, Math and Programming**

**Dylan McNamee**

**Mar 21, 2017**



# CONTENTS

<b>1</b>	<b>Getting started</b>	<b>1</b>
1.1	What you'll need . . . . .	1
1.2	Why cryptography? . . . . .	2
1.3	Let's get started . . . . .	2
1.4	Things to ponder . . . . .	4
1.5	Take-aways . . . . .	4
<b>2</b>	<b>Encoding data into bits</b>	<b>7</b>
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
<b>3</b>	<b>Introducing Cryptol</b>	<b>19</b>
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
<b>4</b>	<b>Implementing More Complex Programs</b>	<b>37</b>
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 . . . . .	44
<b>5</b>	<b>The Enigma</b>	<b>47</b>
5.1	Using a real Enigma . . . . .	48
5.2	Making and using your own Enigma . . . . .	49
5.3	Implementing the Enigma in Cryptol . . . . .	50
<b>6</b>	<b>Completing the Enigma in Cryptol</b>	<b>61</b>
6.1	Combining multiple rotors . . . . .	61
6.2	Creating an Enigma state structure . . . . .	64
6.3	Stepping the rotors . . . . .	65
6.4	Implementing the plugboard . . . . .	68
6.5	Assembling our parts into an Enigma . . . . .	71
6.6	Finishing touches . . . . .	72
6.7	Decoding some real Enigma messages . . . . .	75
	<b>Index</b>	<b>77</b>

## 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.

Cryptography can be used for other purposes than keeping information secret, too. For example, *cryptographic signatures* can be used in place of a pen-and-ink signature to indicate you agree to something. Another application is *cryptographic hashes*, which can tell you, with high confidence, whether a document has been tampered with.

## 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.

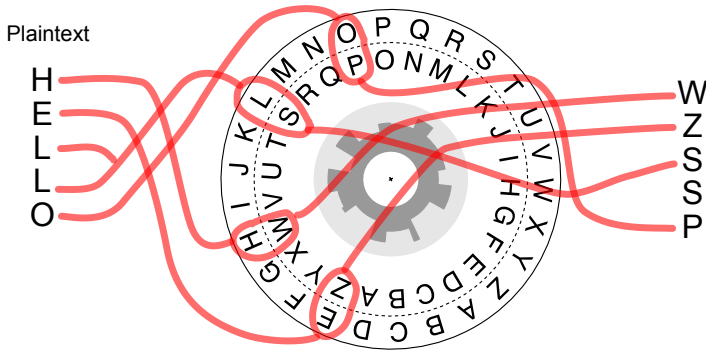


Fig. 1.1: Encrypting “HELLO” with the Caesar cipher. Key is A↔D

Follow the instructions on the kit to decode the following punch-lines:

I wondered why the ball was getting bigger ... (Use the key: J↔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 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 modern cryptography works

later in this book.

### 1.4 Things to ponder

1. Notice that the order of the alphabet is reversed on your Caesar cipher's inner wheel. You may have seen a Caesar cipher whose inner wheel and outer wheel go in the same direction. 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 know<sup>1</sup>? 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 ciphertext in a Caesar 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

---

<sup>1</sup> 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.



have thought about how secure it is, including the aspects of key distribution.



## ENCODING DATA INTO BITS

*“There are 10 kinds of people in the world: those who understand 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 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. Ones 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 on<sup>1</sup>.

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  $n^{\text{th}}$  digit is the  $10^{n-1}$ -place<sup>2</sup>. 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 0s for the values you don't want to add.

---

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

<sup>2</sup> 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..."<sup>3</sup>.

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

---

<sup>3</sup> 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 ones 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 that results in a wrong answer has occurred either before or after the addition has happened. Hint: think about the various possible cases separately.
4. Two's complement is a slight change from *ones' 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 ones 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 can 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” characters in the table, sp is the space character, and del is delete<sup>4</sup>.

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

---

<sup>4</sup> 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 World 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-8<sup>5</sup> 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  $\leq 127$ ), 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!

---

<sup>5</sup> UTF-8 was invented at Bell Labs by Ken Thompson, who co-invented Unix, and Rob Pike, who subsequently moved to Google where he invented the Go programming language (among other accomplishments).

### 2.3.3 An aside: Hexadecimal

Writing numbers in binary is tedious for mere humans<sup>6</sup>. 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 *ones place*, but the next bigger digit in hex is the *sixteens place*<sup>7</sup>, 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 safe<sup>8</sup> 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 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.

---

<sup>6</sup> computers, on the other hand, seem to thrive on tedium.

<sup>7</sup> and the next digit is the 256ths place!

<sup>8</sup> except when 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

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 web page 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 1s 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 *metadata* about the file<sup>9</sup>.

2. **File archives** are encodings that combine a bunch of files and folders into one file that can be sent by email, or downloaded

---

<sup>9</sup> The word *metadata* literally means "data about data", which particularly makes sense in this context.

from a web site, 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  $10_2$  happens to be equal to  $2_{10}$ . Finally you've learned that nerds (the author included) can have a terrible sense of humor.

## 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 web site, 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.

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

```
% cryptol

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

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.

## 3.2 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 in bits 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 number `0xFAB` would be written:

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

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

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



I would call this “a sequence of five elements, each having four bits”. It turns out that the `Bit` at the end is optional: if the last thing in a type is a length (`[4]` for example), you are supposed to add the `Bit` in your understanding of the type.

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 almost exactly the same, 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	<div><div></div><div>f(x,y) = x + y</div><div></div></div>	-----> 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?  $4+4 = 0$ ? 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<sup>1</sup>:

```
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:

---

<sup>1</sup> another way to do this is use decimal numbers, which are friendly, but specify the width of the output, like this: `4 + 4 :[8]`.

```
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  $n^{\text{th}}$  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 6<sup>th</sup> letter of the alphabet. Why the 6<sup>th</sup> character and not the 5<sup>th</sup>? 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 `"cdefghijklmnopqrstuvwxyzab"`. 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” ( $\Rightarrow$ ). 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” ( $\rightarrow$ ). 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  $\Rightarrow$ , 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 ciphertext 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<sup>2</sup>:

```
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"
```

---

<sup>2</sup> 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	a	Z	z
Hex ASCII	0x41	0x61	0x5a	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's ASCII value is less than 0x61, we can add 0x20 to make it lower 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 technology<sup>1</sup>. 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.c` 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 =
    [ encryptChar (codeWheel key) c | c <- message ]
decrypt key message = encrypt key message
```

---

<sup>1</sup> 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.

The first line (`// Caesar cipher`) is a comment: it's text to remind you what's in the file. With our more complex programs, we'll add more comments. They'll help remind us what's going on.

## 4.2.2 Running Cryptol with `caesar.cry`

Now that we have a file with our Caesar cipher in it. To run that program, launch Cryptol from inside your project directory but include the name of the file you want it to load, like this:

```
$ cryptol caesar.cry

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

Loading module Cryptol
Loading module Main

Main> encrypt 11 "using files now"

[warning] at <interactive>:1:1--1:28:
  Defaulting type parameter 'bits'
    of literal or demoted expression
    at <interactive>:1:9--1:10
  to 3
[0x6a, 0x6c, 0x76, 0x71, 0x78, 0x20, 0x79, 0x76,
 0x73, 0x7a, 0x6c, 0x20, 0x71, 0x70, 0x68]
```

Whoops, we still have to turn on ASCII output. If you got `[error] can't find file: caesar.cry` instead, then you need to use a shell command to move into your `CryptoBook` directory. If 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:

```
"wlszknzlosgeyfyueyalknzqlsfmoaosqlhozzob"
```

## 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  $i^{\text{th}}$  character, use the Caesar cipher with the  $i^{\text{th}}$  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 / RESOURCE)

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 `vigenere.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 `vigenere.cry`. Finally, start up Cryptol:

```
$ cryptol vigenere.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:<sup>2</sup>

```
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 .. 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:

---

<sup>2</sup> Note the \s - you can either type this all on one line, or use \s and include new-lines where they appear here.

```
vigenere 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: "thisphraseis-mykey"

"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.



## THE ENIGMA



*Enigma*<sup>1</sup> was the name of a typewriter-like cryptographic machine that was used by Germany before and during World War II. German military commanders used the Enigma to encrypt instructions sent by radio to their commanders across Europe and the Atlantic ocean. The Allies could eavesdrop on the radio transmissions and hear the encrypted messages, but without the key used for that day,

---

<sup>1</sup> Photo by Alessandro Nassiri of the Enigma machine located at the Museo scienza e tecnologia in Milano, Italy, from Wikipedia.

they couldn't decrypt them. A top-secret team of mathematicians were gathered in Bletchley Park, England, with the task of breaking the Enigma. The team's intellectual leader was a young man named Alan Turing. The story of how the Enigma was broken is told pretty well in the movie *The Imitation Game*, and even better (but without Benedict Cumberbatch) in the book *Alan Turing: The Enigma* by Alan Hodges. Alan Turing's life ended tragically due to how homosexual people were treated at the time<sup>2</sup>. In addition to ending WWII early, Alan Turing is credited with inventing the modern computer and artificial intelligence.

In the next two chapters you will learn how to build and use a cardboard model of the Enigma that is compatible with the original. Then, step by step, we'll implement the Enigma in Cryptol, and use it to encode and decode messages.

## 5.1 Using a real Enigma

If you ever get a chance to visit the National Cryptologic Museum<sup>3</sup>, you can use a real Enigma machine. Here's what you would do to encrypt something with a real Enigma machine.

First, the key is the combination of two things: the *rotor settings* and the *plugboard settings*. The rotors are the wheels at the top of the machine. Choose a set of initial positions and write them down. You probably will not be able to change the plugboard settings, but write them down (or take a photo) if you can. After you configure the key, then you start typing your message. As you type, the machinery moves, and lights up letters in the area above the keyboard. As you type the message, a partner writes down the sequences of lit-up letters. This is the encrypted message. As you type, the rotors change positions, so if you want to immediately decode the message, you'll need to reset the rotor, then you start typing in the ciphertext. If

---

<sup>2</sup> The British government issued a formal pardon and apology in 2013.

<sup>3</sup> The National Cryptologic Museum is next to Fort Meade, in Maryland, where the US National Security Agency is located. It's well worth visiting. It's about halfway between Washington D.C. and Baltimore.

things are working right, the plaintext of your message will show up in the lights.

## 5.2 Making and using your own Enigma

Franklin Heath, a cybersecurity company in the UK, released an excellent cardboard model of the Enigma<sup>4</sup>. It is similar to a pen-and-paper version of the Enigma designed by Alan Turing while he was at Bletchley Park, and can be used to encrypt and decrypt messages interchangeably with real Enigma machines.

The first thing you'll need to assemble your own paper Enigma is a tube that has a diameter of 75mm. One source of such tubes is Pringles cans (they call them "crisp tubes" in Britain). If you don't want to buy a can of Pringles, you can make your own tube from cereal box cardboard.

Let's build our paper Enigma step by step, and write Cryptol code that simulates the system at each step.

Following the directions on the Franklin Heath paper Enigma wiki, build the most simple Enigma, which has three paper bands: the Input/Output band, Rotor I, and Reflector B. Make sure to line up the gray bar on the Reflector and the I/O band, and tape those bands stationary, while allowing the rotor to move. The key in this Simple Enigma is the letter between the gray bars when you start. Before decoding each character, slide the rotor toward you one position, so the next higher letter in the alphabet is between the grey bars.

Using this one-rotor setup, set the key to A, and decode the following message:

YMXOVPE

I'll walk you through decoding the first letter:

1. First, advance the rotor so that B is between the grey bars.

---

<sup>4</sup> You can download it from [http://wiki.franklinheath.co.uk/index.php/Enigma/Paper\\_Enigma](http://wiki.franklinheath.co.uk/index.php/Enigma/Paper_Enigma)

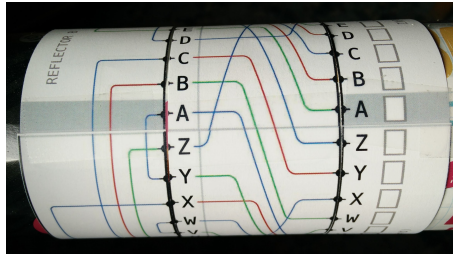


Fig. 5.1: Paper Enigma with one rotor

2. Next, start at the Y on the INPUT/OUTPUT ring on the right.
3. Trace from the Y along the blue line to the J on the rotor.
4. Along the reflector, go from the J along the blue line up to the Q.
5. From the Q, follow the red line across and down to the G.
6. G is the first decoded character! Now advance the rotor so that C is between the grey bars, and decode the next letter.

The result should be two English words. If it isn't, make sure you're advancing the rotor between each character. When you're done, H should be the character between the grey bars.

## 5.3 Implementing the Enigma in Cryptol

The Enigma is way more complicated than the Caesar or Vigenere ciphers. We can't just implement it in one go - instead we have to break it up into the various components and test each step as we go. We'll start by implementing just the rotor, then the reflector, then combine them into a one-rotor Enigma, and so on. Let's go!

### 5.3.1 Implementing Enigma rotors in Cryptol

If you look at what the Enigma rotor does, it takes a letter as input and shifts that letter to another location on the cylinder. The figure below shows a part of Rotor I. If you put the rotor's A between the grey bars, you can trace out what each letter gets transformed to. For example, A from the I/O band goes up four positions to E, and D goes up two positions to F. Because the rotors rotate<sup>5</sup>, it makes sense to think about the rotors in terms of offsets, or *relative* translations instead of letters, or *absolute* addresses, like we did with the Caesar cipher.

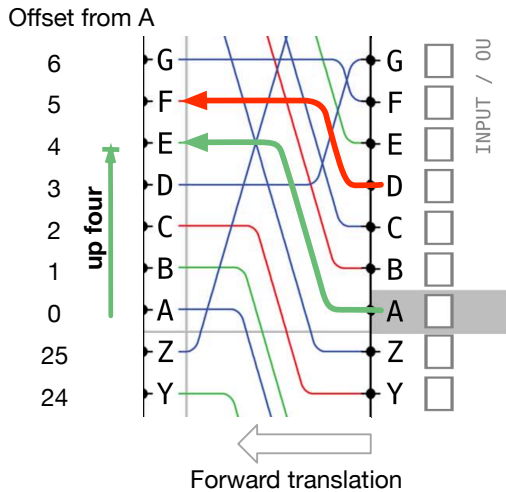


Fig. 5.2: Tracing rotor offsets in the forward direction.

Already we're now talking about three different kinds of numbers: the *ASCII code* for each letter, the *indexes* within the alphabet those letters have, and now *offsets*, which is what we're calling the distance between two letters. They're all numbers, but they have different

<sup>5</sup> Rotors were well-named, weren't they?

meanings, and we need to be careful that we don't mix them up in the code. Here's an example of each of them:

Char	ASCII	Index	Offset from Char →				
			A	B	...	Y	Z
A	65	0	0	1		24	25
B	66	1	25	0		23	24
C	67	2	24	25		22	23
...							
X	88	23	3	4		1	2
Y	89	24	2	3		0	1
Z	90	25	1	2		25	0

The offset representation is how we'll describe what each rotor does in the code. To create the offsets for a rotor it's convenient to trace the lines on our cardboard Enigma and write down which *letter* each line goes to, then write a *helper function* to take that representation and turn it into the sequence of *offsets* we want.

To express this in Cryptol, we start with by tracing the lines for each character in the alphabet. We write them down in the same order, so the first character is what the line from A goes to, which is E. B's line goes further up, to K, and so on. In your `enigma.cry` file, create a variable called `rotorIchars` like this:

```
rotorIchars = "EK ..."
```

Where you fill in the rest of the string.

Our next job is to write a helper function that computes each offset. We also choose to represent offsets with a different number of bits than characters, so we (and Cryptol) can tell them apart from each other. We *could* use 8 bits for characters, indexes and offsets, but using different numbers of bits for each of them makes it easier for Cryptol to help prevent us from getting them confused, by giving us an error when we provide one kind of number when a function expects another.

```
type Char      = [8] // ASCII characters
type Index     = [7] // A -> 0, .. Z -> 25
```



```

type Offset    = [6] // distance between 2 chars

alphabet       = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
// rotorIchars = "EK ... " // <- from your Enigma

// some helpers to go from Char to Index and back:
indexToChar : Index -> Char
indexToChar i = (0b0 # i) + 'A'

charToIndex : Char -> Index
charToIndex c = drop`{1}(c - 'A')

stringToOffsets : [26]Char -> [26]Offset
stringToOffsets chars =
  [ drop`{2}((c + 26 - a) %26)
    | c <- chars
    | a <- alphabet
  ]

```

The main tricky bit in this part of the program is that we need to avoid negative numbers, because they won't work with Cryptol's modulo arithmetic<sup>6</sup>. Fortunately, that's easy to do. Let's work through how to compute offsets using only positive numbers. First, as we're marching through the chars sequence, the offset between each character *c* and its position in the alphabet *a* is *c* - *a*. This is a positive number when the line goes up on our rotor. For example, computing the very first entry, 'E' - 'A' equals 4, just like we'd hope.

However, looking up the rotor at G, we get 'D' - 'G' which is -3. How do we fix this? Well, it's always safe to add 0 to something – doing that doesn't change the result, right? In modulo arithmetic, there are a *bunch* of numbers whose values are 0. For example, 26 % 26 is 0, and so is 52 % 26. So if we add 26 to our element of chars before subtracting, that's the same as adding zero, and we also know that the result will be positive.

So, our final expression for computing the offset from our array is *c*

<sup>6</sup> if you type `-3 % 26` in Cryptol, you get 3 instead of 23 like we'd hope. Python gets this right, but C, Java and many other languages get it wrong (they usually say -3). It's always safe to keep things positive like we're doing here.

+ 26 - a. We do drop `{2}` of that to take the 8 bit number and drop the two leading 0's from it. Think about whether it's always safe to assume there will be at least two leading 0's in this number.

To test your function, try loading your program and running it, like this:

```
% cryptol enigma.cry
... (Cryptol talking to you)
Main> :set base=10
Main> stringToOffsets rotorIchars
[4, 9, 10, 2, 7, 1, 23, ... ]
```

If you get a different sequence of numbers, first decide if you agree with the offsets above, and if you do, figure out why your function doesn't do the same thing.

### 5.3.2 Exercise: write `applyOffsetToIndex`

To apply an offset to an index  $i$ , we use this recipe: first, look up the offset in the offset sequence using the `@ i` indexing operator. To add two numbers, they need to have the same number of bits, so prepend a `0b0` to the front of the offset using the `#` operator. Then we add the result to  $i$  and apply `% 26` to get the new index.

Following this recipe, write a function called `applyOffsetToIndex` that has the following type:

```
applyOffsetToIndex : [26]Offset -> Index -> Index
applyOffsetToIndex offsets i = <fill this in...>

Main> let rIoffs = stringToOffsets rotorIchars
Main> applyOffsetToIndex rIoffs (charToIndex 'A')
Main> applyOffsetToIndex rIoffs (charToIndex 'Z')
Main> applyOffsetToIndex rIoffs (charToIndex 'Y')
```

Does your code agree with the cardboard Enigma?

### 5.3.3 Implementing the reflector in Cryptol

Now let's look at the Reflector. The main thing different between this and the rotor is that the lines loop back to the edge they start from.

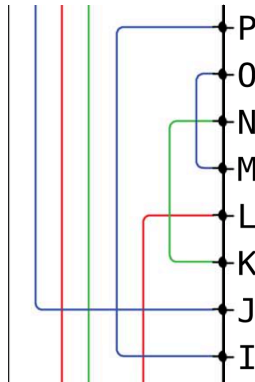


Fig. 5.3: A portion of Reflector B.

Here, we see that I goes to P, K goes to N and so on. Just as we did for the rotor, follow the lines and come up with the Cryptol string that represents the Reflector's actions. It should start like this:

```
reflectorBchars = "YRU // ... finish the rest"
```

We can now reuse our code from the rotors to compute the offsets in the reflector, like this:

```
reflectorBoff = stringToOffsets reflectorBchars
```

Now test your reflector and your `applyOffsetToIndex` function:

```
Main> indexToChar (applyOffsetToIndex reflectorBoff_
↪(charToIndex 'P'))
'I'
Main> indexToChar (applyOffsetToIndex reflectorBoff_
↪(charToIndex 'O'))
```

```
'M'  
Main> indexToChar (applyOffsetToIndex reflectorBoff_  
↪ (charToIndex 'K'))  
'N'
```

Looking at the figure, indeed  $P \rightarrow I$ ,  $O \rightarrow M$  and  $K \rightarrow N$ . Go ahead and test it on A, B and C and compare it with your Enigma to increase your confidence.

### 5.3.4 Going reverse through the rotor

Because of the way the reflector works, if  $I \rightarrow P$ , we know the reverse is also true:  $P \rightarrow I$ . This kind of transformation is called a *self-inverting function*. You may have already noticed that the rotors are *not* self-inverting. Looking at Rotor I, going in the forward direction,  $A \rightarrow E$ , but going from left-to-right, E goes off the top of the figure to L. So towards the goal of implementing a one-rotor Enigma, we're 2/3rds of the way there: we can go forward through the rotor, then through the reflector, and now what we need to do is go backwards through the rotor. Since the rotor is *not* self-inverting, we'll have to compute the backwards function.

We could go through, one by one, and produce another string that represents the backwards transformation. However, we already have the information we want in the previous rotorIchars string. Look at this:

```
alphabet    = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  
rotorIchars = "EKMFLGDQVZNTOWYHXUSPAIBRCJ"  
           ^- shows E -> A      ^- shows A -> U
```

In the forward direction, this shows us  $A \rightarrow E$ . It also tells us the backwards-mapping too: because E is in the first position, that tells us that in the reverse direction  $E \rightarrow A$ . Because K is in the second position, we know  $K \rightarrow B$ . We can follow this pattern to automate the process of reversing this operation in Cryptol! It's a bit tricky, so we'll go carefully:

```

1 index0f c permutation = candidates ! 0 where
2   candidates = [ -1 ] # [ if c == s then i else p
3                         | s <- permutation
4                         | p <- candidates
5                         | i <- [ 0 .. 25 ]
6                         ]

```

The first function we want is one that gives us the index of a character in a permutation. Line 1 defines our function, and says that it returns the last item of a sequence called `candidates`. The `where` says we're about to define some variables (in this case only one). Line 2 says that `candidates` is a sequence that starts off by concatenating the sequence of one element (`[-1]`) with a *sequence comprehension* (remember those from Chapter 3?). Each element of the sequence is the result of an if statement: if `c == s` it's `i` otherwise it's `p`. We don't yet know what any of those variables (except `c`) is yet, but fear not: they're defined right below. Line 3 says that `s` is drawn from the elements of permutation. So each time through the loop, `s` is the next element of the permutation. Line 4 says that `p` is drawn from the elements of the `candidates` sequence. Interesting: We're using the sequence in the definition of itself! Just like in Chapter 3, this is an instance of *recursion*. Finally, line 5 says that `i` is drawn from the sequence `[0..25]`.

When this function runs, it builds up the `candidates` sequence, starting with `-1`, each element keeps being set to `p` (which starts out with `-1`) until the letter from permutation being examined, called `s` is equal to `c`, the letter we're searching for. When that happens, the new element of `candidates` gets set to `i`, which is the index of the match, because the numbers `0 .. 25` are the indexes of the elements of shuffled sequence.

Here are the values of `candidates` as it proceeds through the string, with the call `findIndex 'L' rotorI`:

```

c: 'L'
i:      [ 0,  1,  2,  3,  4,  5,  6, .., 25 ]
candidates = [-1, -1, -1, -1,  4,  4,  4, .., 4 ]
s:      E  K  M  F  L  G  D  ... J

```

```
note:          s == 'L' here: ^   so the index i
              (4) is saved to candidates
```

With this function, we can create the left-to-right version of a rotor given its right-to-left version:

```
invertPermutation perm =
  [ indexToChar (indexOf c perm)
  | c <- alphabet
  ]
```

Save these functions and the definition of `rotorIchars`, `reflectorBchars` and `alphabet` to a file called `enigma.cry`, and run `Cryptol` on it:

```
$ cryptol enigma.cry

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

Loading module Cryptol
Loading module Main
Main> :set ascii=on
Main> let rotorIrev = invertPermutation rotorIchars
Assuming a = 7
"UWYGADFPVZBECKMTHXSLRINQOJ"
Main> rotorIrev @ asciiToIndex 'C'
'Y'
Main> invertPermutation reflBchars == reflBchars
True
```

Indeed, going from left-to-right (backwards), C goes to Y. Pretty cool, isn't it? We worked hard to write this code to save us the hassle of manually tracing the letters backwards. The benefit of doing it this way instead of by hand is that we have confidence that the backwards version of the rotors is actually correct. A single typo in the string would result in an error that would be really hard to track down.

Finally, at the end, we tested whether the reflector is its own inverse permutation, and indeed it is.

### 5.3.5 Combining the Rotor and Reflector

Now we can combine the functions you've written so far into an implementation of a one-rotor Enigma:

```

1  // encryptOneChar
2  // takes the offsets of a rotor, its inversion,
3  // a reflector and a character and
4  // returns the encrypted character
5  encryptOneChar : [26]Offset -> [26]Offset ->
6                  [26]Offset -> Char -> Char
7  encryptOneChar rotorFwd rotorRev reflector
8                  inputChar = outputChar where
9      inputIndex    = charToIndex inputChar
10     afterRI       = applyOffsetToIndex rotorFwd inputIndex
11     afterRefl     = applyOffsetToIndex reflector afterRI
12     outputIndex   = applyOffsetToIndex rotorRev afterRefl
13     outputChar    = indexToChar outputIndex
14
15 // encryptOneRotor
16 // takes the strings for a rotor and a reflector
17 // and message and encrypts the message one character
18 // at a time, first rotating the rotor each step
19 encryptOneRotor rotor reflector message =
20     [ encryptOneChar (rotorOff <<< i)
21         (rotorRevOff <<< i)
22         reflectorOff c
23     | c <- message
24     | i <- [1 .. 100]
25     ] where
26     rotorOff = stringToOffsets rotor
27     rotorRev = invertPermutation rotor
28     rotorRevOff = stringToOffsets rotorRev
29     reflectorOff = stringToOffsets reflector

```

There's a lot of code here, so let's go through it line by line: Lines 5

and 6 define the type of `encryptOneChar`, and lines 7 and 8 define the name of the function and its arguments. We say here that the output of the function is `outputChar`, but the `where` means we're defining `outputChar` and other variables in the following indented lines. Line 9 defines `inputIndex` to be the index of our input character (e.g., A would be 0). Next we define `afterRI` to be the index of the character after passing through Rotor I. Similarly, `afterRef1` is the index after passing through the Reflector. Finally, `outputIndex` is the index after going through `rotorRev`. Finally, we convert `outputIndex` into our `outputChar` by using the well-named `indexToChar` function.

Moving down to the `encryptOneRotor` function, it returns a sequence comprehension, which applies `encryptOneChar` to all of the characters of `message`. In parallel with those characters, we have an index variable `i` go from 1 .. 100, and we rotate the `rotorOff` and `rotorRevOff` by that many steps, to simulate how we rotate the rotors before encrypting each character. The `where` at the end of the comprehension says we define the various sequences afterwards. The most tricky bit here is that we compute the `rotorRev` from `rotor` by using the `invertPermutation` function we wrote earlier. The rest of the variables are just the result of applying `stringToOffsets` to create the offset versions of the rotors, which are what our `encryptOneChar` function needs.

Now we can test it with the exercise from Section 5.2:

```
Main> encryptOneRotor rotorIchars reflBchars "YMXOVPE"  
Assuming a = 7  
"G00DJ0B"
```

Pretty amazing! One limitation of this implementation is that it can only handle messages up to 100 characters long. That, and it's missing a few features from our paper Enigma. We'll take care of those in the next chapter.



## COMPLETING THE ENIGMA IN CRYPTOL

Compared to our paper Enigma (and the real thing), our Cryptol version lacks:

1. Multiple rotors,
2. *pins* that control stepping the rotors,
3. the *rings* that control where the pins start, and
4. the *plugboard*, which modifies the I/O ring.

Let's implement them!

### 6.1 Combining multiple rotors

Get our your paper Enigma and set Rotor II and Rotor III to A. Then for each of them, trace, from the right towards the left, each letter from A .. Z just like we did for Rotor I in the previous chapter. Starting with Rotor II (the middle one), you'll see that the line from A goes straight across to A, so the first letter in that rotor is just A.

As a starting point, here are the first few entries for rotors II and III:

```
rotorIIchars = "AJD ..."  
rotorIIIchars = "BDF ..."
```

Once you complete the rotor strings, the next job is to combine them in the encryption function. One annoying thing that you may have noticed in our `encryptOneRotor` function is when we had to individually rotate the `rotorOff` and `rotorRevOff` sequences. Let's create a data structure that lets us deal with an entire rotor (and its reverse) at the same time:

```
// a rotor elt is: (Fwd, Rev, Pin)
type RotorElement = (Offset, Offset, Bit)
type Rotor = [26]RotorElement
```

This declares `RotorElement` to be a *tuple*, which is a comma-separated list of items inside parentheses. Unlike a sequence, whose elements all have to be the same type, the elements of a tuple can have different types—in this case, `Offset` and `Bit`. The `Bit` at the end is for the Rings, which we'll deal with in the next section. To create a rotor from the characters we carefully traced, we need a helper function, `buildRotor`:

```
buildRotor : [26]Char -> [26]Bit -> Rotor
buildRotor rotorChars pins = [ (of, ob, p)
                              | of <- fwd
                              | ob <- rev
                              | p <- pins
                              ] where
    fwd      = stringToOffsets rotorChars
    revChars = invertPermutation rotorChars
    rev      = stringToOffsets revChars
```

Returning to how the multi-rotor Enigma works, and thinking about what we've written so far, our one-rotor Enigma is the left-hand half of the 3-ring Enigma: the Reflector and first Rotor. We need to add two more rotors to it. It's not quite as simple as that, though, because our `encryptOneRotor` function takes a `Char` as input, and returns a `Char` as output, but if we want to connect our rotors together, the type of data that flows between them is `Index`, not `Char`. So we'll need to *refactor* our code a bit so that it's more modular.

```
doReflector : [26]Offset -> Index -> Index
doReflector rf i = (i + (0b0 # (rf@i)) ) % 26

doRotorFwd : Rotor -> Index -> Index
doRotorFwd r i = (i + (0b0 # (r@i).0 )) % 26

doRotorRev : Rotor -> Index -> Index
doRotorRev r i = (i + (0b0 # (r@i).1 )) % 26

doOneRotor refl rotor i =
    doRotorRev rotor (doReflector refl (doRotorFwd rotor i))

doTwoRotors refl r1 r2 i =
    doRotorRev r2 (doOneRotor refl r1
        (doRotorFwd r2 i))
```

Let's start with our function that implements one rotor and the reflector. If we want to add another rotor to our Enigma, we can call our `doOneRotor` function as a helper. First we go through our new rotor forwards, then call the `doOneRotor` function, then use the output of that, and go through our rotor backwards.

### 6.1.1 Exercise: write the three rotor function

Follow the pattern established above, and write `doThreeRotors`. It should start off like this:

```
doThreeRotors refl r1 r2 r3 c =
```

Make sure you use `doTwoRotors`, and keep track of how many arguments it takes and what they are.

We can wrap this function with another function that translates a `Char` to `Index` and back again so that we can test our code:

```
doOneChar c refl r1 r2 r3 =
    indexToChar ( doThreeRotors refl r1 r2 r3 (charToIndex c))
```

Now we can encode single characters with the three rotor Enigma code, and compare it to our cardboard Enigma. As a test, set all three rotors to A, and make sure that encoding A results in U, and B goes to E. If you don't get those results, or if your program loads with errors, see if you can figure out what's wrong with your version.

```
Enigma> doOneChar 'A' reflB rotorI rotorII rotorIII
'U'
Enigma> doOneChar 'B' reflB rotorI rotorII rotorIII
'E'
```

## 6.2 Creating an Enigma state structure

Since the Enigma state consists of a number of different things, it's going to be convenient to create a *record* to describe it. A record is like a tuple but with named elements. Here's the declaration of our Enigma state record:

```
type EnigmaState = {
  rotors      : [3]Rotor,
  reflector   : Reflector,
  plugboard   : CharPerm
}

buildEnigmaState r1 r2 r3 refl plug =
  { rotors = [r1,r2,r3]
  , reflector = refl
  , plugboard = plug
  }
```

The EnigmaState record type combines all three rotors and the reflector into a single *object*. If we had a variable of this type named `enigmaState`, we could access its rotors with this syntax: `enigmaState.rotors`. Similarly for `plugboard`. The part about the `plugboard` will be explained near the end of this chapter.

## 6.3 Stepping the rotors

Now that we have all three rotors going, our next step is to advance them before encoding each character. Let's start by describing in a bit more detail what the *advance rotors* function needs to do:

First, there are pins attached to each rotor. We're representing the pins as the `Bit` in the `Rotor` data type. When the pin at position 0 of the rotor is `True`, we need to follow these rules:

- If the middle rotor's pin is `True`, we advance all three rotors, otherwise
- if the pin furthest from the reflector (rotor 3) is `True` we advance rotors 2 and 3, otherwise
- just advance rotor 3.

Here's code that implements these rules, along with the helper function `getPins`:

```
// getRotations takes 3 rotors, returns a bit
// for each rotor, whether it should advance.
getRotations : [3]Rotor -> [3]Bit
getRotations rs = [r0, r1, r2] where
    pins = getPins rs
    r0 = pins@1
    r1 = pins@2 || pins@1
    r2 = True

// getPins: Extracts the pin values from 3 rotors
getPins : [3]Rotor -> [3]Bit
getPins rs = [ ((rs@0)@0).2
               , ((rs@1)@0).2
               , ((rs@2)@0).2
               ]
```

The function `getPins` has some tricky indexing, so here's an explanation, step by step:

1. `rs@0` is the first rotor (type: `[26] (Char, Bit)`)

2.  $(rs@0)@0$  is the first element of the first rotor, type:  $(Char, Bit)$
3.  $((rs@0)@0).0$  is the character part of that first element, type:  $Char$
4.  $((rs@0)@0).1$  is the Pin part of that first element, type:  $Bit$

Now we can write `advanceRotors` that applies these rules to our rotors:

```
advanceRotor : Rotor -> Bit -> Rotor
advanceRotor r b = if b then r <<< 1 else r

advanceRotors : EnigmaState -> EnigmaState
advanceRotors state =
  { rotors = newRotors
  , reflector = state.reflector
  , plugboard = state.plugboard
  } where
    newRotors = [ advanceRotor r b
                  | r <- state.rotors
                  | b <- rbits
                  ]
    rbits = getRotations state.rotors
```

Before we can test the function, we need to have some pins in our rotors. Looking at the cardboard Enigma, each rotor's pin is at the position where the letter has a grey background. So Rotor I's pin is at 'Q', Rotor II's pin is at 'E' and Rotor III's pin is at 'V'. So all we need to do is pass to `buildRotor` a sequence of bit's with the bit set accordingly. Here's a helper function that will make that easy:

```
setBit : {a, b} (fin a, fin b, b >= 1)
      => [a] -> [b] -> [b]
setBit b n = n || (1 >>> 1 >> b)

pinsI, pinsII, pinsIII : [26]
pinsI  = setBit (indexOf 'Q' alphabet) zero
pinsII = setBit (indexOf 'E' alphabet) zero
pinsIII = setBit (indexOf 'V' alphabet) zero
```

```
rotorI  = buildRotor rotorIchars pinsI
rotorII = buildRotor rotorIIchars pinsII
rotorIII = buildRotor rotorIIIchars pinsIII
```

You might be surprised by the definition of `setBit`. Our goal is to take a 1 in position 0, and use the *shift* operation (`>> b`), to move it `b` spots to the right. So we first need a 1 in the first position. There are many ways to achieve this<sup>1</sup>. The way this function does it is by taking a 1 in the least significant position, and rotating it one position to the right (`>>> 1`), which wraps it around to the most significant place, which we then shift `b` positions to the right. Finally we use the *or* operator, `|`, to overlay our newly set bit with our input `n`.

The other interesting thing about this code is the use of zero. It's a built-in Cryptol `0` constant that can have any type. In this case, it's a sequence of as many `False` bits as we need to initialize our empty pins.

Now we have enough functions to build a sequence of rotor positions that follow the Enigma's rules:

```
enigmaStates start = estates where
  estates = [start] # [advanceRotors prev | prev <- estates]
```

### 6.3.1 Exercise

You can now build most of an Enigma state, including rotors with the pins. We can feed that starting state to `enigmaStates` to get an infinite sequence of states, which are the states after following the `advanceRotors` rules for each step. If drop the first 99 states of that sequence, you'll see the sequence of states just before and after a pin shows up in position 0 of Rotor II, so run this:

---

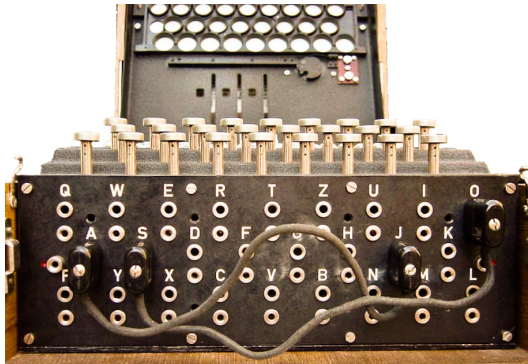
<sup>1</sup> Another way to achieve it would be to write `(0b1 # zero)`. Since Cryptol knows how many bits the result needs to have, it will stretch zero to have the right number of 0 bits after the 0b1 to work out.

```
Enigma> let startState = buildEnigmaState rotorI rotorII
↳rotorIII reflB zero
Enigma> drop`{99} (enigmaStates startState)
[{rotors = [(0x04, 0x14, False), (0x09, 0x15, False),
...

```

Explain how the sequence of Rotor states do, or do not, follow the rules for rotor advancement.

## 6.4 Implementing the plugboard



Here's how the plugboard works on the cardboard Enigma: to configure the plugboard, you're given pairs of letters, like "AP" and "BR". In the little boxes on the I/O Ring, you write a P next to the A and write an A next to the P, and so on. Then, when you're about to encode a letter, you first check if there's a letter in the box, and if there is, you hop over to the corresponding letter on the ring, and start encoding from there. Similarly on the way back after the reflector and the rotors, you return the letter in the box (if there is one) when you finish following the lines.

It's easiest to construct the plugboard as a string that starts out as the alphabet in order (which is the same as a *empty plugboard*), and for



each pair of letters in the plugboard's definition, swap those letters in the string. Here's a helper function that does that:

```
buildPlugboard : {n} (fin n, width n >= 1) =>
  Swaps n -> [26][8] -> [width n] -> CharPerm

buildPlugboard swaps permutation i =
  if i >= width swaps
  then permutation
  else buildPlugboard swaps swappedPerm (i + 1) where
    swappedPerm = doSwap (swaps@i) permutation

doSwap : [2][8] -> [26][8] -> [26][8]
doSwap [c, d] perm = perm2 where
  index0fC = index0f c perm
  index0fD = index0f d perm
  perm1 = [ if i == index0fC then d else x
            | x <- perm | i <- [0..25]]
  perm2 = [ if i == index0fD then c else x
            | x <- perm1 | i <- [0..25]]

franklinHeathPlugboard = buildPlugboard
  ["AP","BR","CM","FZ","GJ","IL","NT","OV","QS","WX"]
  alphabet 0
```

This function uses a different kind of recursion than we've used before. Rather than recursion based on elements of a sequence, we're calling our `buildPlugboard` function in its own definition, but with a bigger value for `i` each time around. Finally it ends when `i` is greater than or equal to the number of elements in our swaps sequence. Cryptol's built-in `width` function tells us how many elements there are in swaps.

Finally, to use the plugboard, even though its operation is simple, it's also subtle: in the forward direction, its input is a `Char`, but its output is an `Index`. In the reverse direction, its input is an `Index`, but its output is a `Char`.

### 6.4.1 Exercise: Testing the plugboard

Implement the `doPlugFwd` and `doPlugRev` functions, which have these types:

```
doPlugFwd : CharPerm -> Char -> Index
```

```
doPlugRev : CharPerm -> Index -> Char
```

It's subtle, but all you should need are the *indexing operator* `@` and our helper function from before, `charToIndex`.

As an aside, you've probably noticed the Cryptol prompt always says `Main>`. If you want, you can start your `enigma.cry` file with the following line, which will change the prompt to `Enigma>`, which is pleasing (to me, at least):

```
module Enigma where
```

When you're done, test your function with an empty plugboard, as well as with the `cardboardEnigma` example plugboard, which is defined by this sequence of character pairs:

```
franklinHeathPlugboard = buildPlugboard
  ["AP", "BR", "CM", "FZ", "GJ",
   "IL", "NT", "OV", "QS", "WX"]
  alphabet 0

// save the above to your enigma.cry,
// and then in Cryptol:

Enigma> set ascii=off
Enigma> doPlugFwd franklinHeathPlugboard 'A'
0x0f
Enigma> doPlugFwd franklinHeathPlugboard 'X'
0x16
Enigma> set ascii=on
Enigma> doPlugRev franklinHeathPlugboard 0
'P'
Enigma> doPlugRev franklinHeathPlugboard 25
```

```
'F'
```

## 6.5 Assembling our parts into an Enigma

We now have the components required to write a function that encrypts a single character, by going through the plugboard, then the three rotors and reflector, and back through the plugboard:

```
doOneChar : Char -> EnigmaState -> Char
doOneChar c es =
    doPlugRev es.plugboard
        (doThreeRotors
            es.reflector
            es.rotors
            (doPlugFwd es.plugboard c))
```

And now we can use our well-worn technique of constructing a sequence of encrypted characters by applying `doOneChar` to each character of the input, in parallel with each state of the Enigma machine:

```
doEnigma startState message =
    [ doOneChar c es
    | c <- message
    | es <- drop`{1}`(enigmaStates startState)
    ]
```

The reason we drop the first state of our `enigmaStates` is that when you use the Enigma, you're supposed to advance the rotors *before* encrypting each character.

### 6.5.1 Exercise: Testing the plugboard, three rotors and reflector

We can now test the basic functions of the Enigma. First, we'll create a start state corresponding to the initial rotor positions AAA:

```
aaaEnigmaState = buildEnigmaState
    rotorI rotorII rotorIII reflB
    franklinHeathPlugboard

// and another state with an empty plugboard:
noPlugboardState = buildEnigmaState
    rotorI rotorII rotorIII reflB
    alphabet
```

Then, in Cryptol, we can encrypt and decrypt a test message:

```
Enigma> doEnigma aaaEnigmaState "WOWITWORKS"
"JLBDMQCLQZ"
Enigma> doEnigma aaaEnigmaState "JLBDMQCLQZ"
"WOWITWORKS"
```

If it works, congratulations! If it doesn't, it's most likely your plugboard, and you can test things with an empty plugboard like this:

```
Enigma> doEnigma noPlugboardState "WOWITWORKS"
"KIYQXAMTSO"
```

If that doesn't work, manually check your `doOneRotor` and `doTwoRotor` functions against your cardboard Enigma, to make sure things are working that far.

## 6.6 Finishing touches

All we have left before our Enigma is complete is to implement adjustable rings, and being able to set the initial rotor positions according to a three-letter *key*. Let's do it!

### 6.6.1 Implementing adjustable rings

Install the rings on your cardboard Enigma. Some descriptions of the Enigma describe a ring position by which letter on the ring covers up

the A on the rotor underneath. Other descriptions specify the index<sup>2</sup> of the character covering the rotor's A. Fortunately, it's easy for us to do either. Let's look at a very simple ring adjustment: set Rotor III's ring so that the ring's A covers the rotor's B. Look at what they do, and think about how to implement that in your program. Here are some attributes of the rings that you might notice:

1. They repeat the alphabet that's on the rotor underneath,
2. which *character* has a gray background is fixed, per rotor, and
3. when you slide the ring around, the characters and the pin move relative to the lines on the rotor.

From these observations, it that setting the rings should consist of rotating the offsets and the pins in one direction or the other. Let's figure out which direction the offsets get rotated according to a ring position.

Start by creating a new version of `buildRotor` that includes a `ring` parameter:

```
buildRotorRing rChars pins ring = [ (of, ob, p)
                                     | of <- fwd
                                     | ob <- rev
                                     | p <- pins
                                     ] where
fwd      = (stringToOffsets rChars) ??? ring
revChars = invertPermutation rChars
rev      = (stringToOffsets revChars) ??? ring
//
//                                     ^^^
//                                     should these be >>> or <<<?
```

To figure out which direction to rotate the `fwd` and `rev` sequences, try it with our cardboard Enigma: set the ring's B to cover the rotor's A, and align the ring's A with the I/O ring's A. See what happens to the translation, compared to before.

---

<sup>2</sup> Usually instead of specifying the 0-based index, they specify the 1-based "which letter of the alphabet" ( $A \rightarrow 1$ ), which is just the *index plus 1*. Which is also easy for us to compute.

If that helps you enough to decide, go ahead and complete and add this function to your `enigma.cry`. If you need more help, try comparing the output of `doRotorFwd`, with rotors constructed both ways, to what your cardboard Enigma does.

Once you've added your version of `buildRotorRing`, you can test it by building an `enigmaState` that uses those rotors, like this:

```
rotorI   = buildRotorRing rotorIchars pinsI (charToIndex 'J')
rotorII  = buildRotorRing rotorIIchars pinsII (charToIndex 'N')
rotorIII = buildRotorRing rotorIIIchars pinsIII (charToIndex 'U'
↳ ')

testState = buildEnigmaState
            rotorI rotorII rotorIII reflB
            franklinHeathPlugboard
```

Then in Cryptol, run:

```
Enigma> :set ascii=on
Enigma> doEnigma testState "ALMOSTDONE"
"YXRJLPTZWS"
```

If you don't get that result, check the output of one rotor:

```
Enigma> :set ascii=off
Enigma> doRotorFwd rotorIII 0
0x01
```

### 6.6.2 Setting the key

The configuration of the Enigma corresponding to a *key* consists of rotating each rotor in the start state so that the key's letter is in position 0. We can pretty easily do that, again using the rotate operator (`<<<`), an amount corresponding to the difference between the key's index and 'A'. So, let's write a function that takes an Enigma state and a key (`[3] Char`), and produces a new Enigma state with the rotors set appropriately:

```
setKey : EnigmaState -> [3]Char -> EnigmaState
setKey state key = { rotors      = rs'
                    , reflector = state.reflector
                    , plugboard = state.plugboard
                    } where
  rs' = [ r <<< 0
        | r <- state.rotors
        | o <- keyOffs ]
  keyOffs = [ charToIndex c | c <- key ]
```

## 6.7 Decoding some real Enigma messages

Reviewing what we've done:

1. we can build an `EnigmaState` corresponding to a sequence of rings, a reflector and plugboard using `buildEnigmaState`,
2. to change the key of a state from "AAA" to something else, we use `setKey`, which takes a state and a key, and returns a new state.

The Franklin Heath website has an example Enigma message that has been encoded with the Ring and Plugboard we've already created. All that's left is to create the state and set the key, which you can do in your `enigma.cry` or at the Cryptol command line, like this:

```
Enigma> let state = buildEnigmaState rotorI rotorII rotorIII_
↳ reflB franklinHeathPlugboard
Enigma> let vqqKey = setKey state "VQQ"
Enigma> doEnigma vqqKey "HABHVHLYDFNADZY"
```

What do you get? It should be four words smooshed together. If that's not what you get, go back through the various checkpoints in this chapter to make sure you have the correct setup.

If it did work, try decrypting the following message, with the same key as before: `FONUGETQBLRQKHFE SDSUFTEAHVZP`

Finally, if you've been working with a partner, each one of you come up with a message (and maybe even a different key, plugboard, or rotor orders), and describe the configuration to them sufficiently that they can decode your message.

You may note that, just as in Chapter 1, the challenge with secure communication has been transformed into a *key distribution* challenge.



# INDEX

## A

Alan Turing, 47  
algorithm, 19  
archives, 17  
arguments, 26  
arguments vs. parameters, 26  
ASCII, 12

## B

binary, 8  
binary addition, 9  
bitmaps, 17  
Bletchley Park, 47

## C

conditional statements, 32  
Cryptol, 19

## D

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

## E

encoding, 7  
Enigma, 47  
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, 4  
key expansion, 41

## L

lazy evaluation, 43

## M

metadata, 17  
modulo arithmetic, 53

## N

number bases, 7

## O

ones' complement, 11

or operator, 67  
overflow, 11

### P

parallel comprehension, 42  
parameters, 23, 26  
performance, 11  
permutation, 57  
pixels, 17

### R

record, 64  
recursion, 41, 57, 69  
refactor, 62  
reverse index operator, 25  
rotate operator (>>>), 29  
rpm file, 17

### S

self-inverting function, 56  
sequence, 21  
sequence comprehension, 57

### T

tar file, 17  
text editors, 37  
text file, 37  
transistors, 11  
tuple, 62  
two's complement, 10  
type, 20

### U

Unicode, 13  
UTF-8, 13

### V

variable, 22  
vector graphics, 17  
Vigenère cipher, 40

### W

where, 57

World War II, 47

### Z

zero, 67  
zero-based indexing, 25  
zip file, 17