Programming Cryptol

Cryptol:

The Language of Cryptography

| galois |

Galois, Inc.
421 SW 6th Ave | Suite 300 | Portland, OR 97204
T: 503.626.6616 | F: 503.350.0833
www.galois.com

IMPORTANT NOTICE

TRADEMARKS

Galois, Inc.
421 SW Sixth Avenue, Suite 300
Portland, OR 97204

Contents

Contents **iii**

1 A Crash Course in Cryptol **1**

1.1 Basic data types . . . 1
1.2 Bits: Booleans . . . 1
1.3 Words: Numbers . . . 2
1.4 Integers: Unbounded numbers . . . 2
1.5 Rationals . . . 3
1.6 Floating Point Numbers . . . 3
1.7 Tuples: Heterogeneous collections . . . 4
1.8 Sequences: Homogeneous collections . . . 4
1.8.1 Enumerations . . . 5
1.8.2 Comprehensions . . . 5
1.8.3 Appending and indexing . . . 6
1.8.4 Finite and infinite sequences . . . 6
1.8.5 Manipulating sequences . . . 7
1.8.6 Shifts and rotates . . . 8
1.9 Words as sequences . . . 8
1.10 Characters and strings . . . 9
1.11 Records: Named collections . . . 10
1.12 The `zero` . . . 10
1.13 Arithmetic . . . 11
1.14 Types . . . 13
1.14.1 Monomorphic types . . . 13
1.14.2 Polymorphic types . . . 14
1.14.3 Predicates . . . 15
1.14.4 Why typed? . . . 16
1.15 Defining functions . . . 16
1.15.1 Definitions in the interpreter with `let` . . . 17
1.15.2 Local names: `where` clauses . . . 17
1.15.3 λ-expressions . . . 18
1.15.4 Using `zero` in functions . . . 18
1.16 Recursion and recurrences . . . 18
1.17 Stream equations . . . 21
1.18 Type synonyms . . . 21
1.19 Type classes . . . 22
1.20 Type vs. value variables . . . 23
1.20.1 Positional vs. named type arguments . . . 24
1.20.2 Type context vs. variable context . . . 24
1.20.3 Inline argument type declarations . . . 25
1.21 Type constraint synonyms . . . 25
1.22 Program structure with modules . . . 25

1.23 The road ahead . . . 26

2 Classic ciphers **27**
2.1 Caesar's cipher . . . 27
2.2 Vigenère cipher . . . 28
2.3 The atbash . . . 29
2.4 Substitution ciphers . . . 29
2.5 The scytale . . . 30

3 The Enigma machine **33**
3.1 The plugboard . . . 33
3.2 Scrambler rotors . . . 33
3.3 Connecting the rotors: notches in action . . . 34
3.4 The reflector . . . 36
3.5 Putting the pieces together . . . 36
3.6 The state of the machine . . . 37
3.7 Encryption and decryption . . . 38

4 High-assurance programming **41**
4.1 Writing properties . . . 41
4.1.1 Property–function correspondence . . . 42
4.1.2 Capturing test vectors . . . 42
4.1.3 Polymorphic properties . . . 42
4.2 Establishing correctness . . . 43
4.2.1 Formal proofs . . . 44
4.2.2 Counterexamples . . . 44
4.2.3 Dealing with polymorphism . . . 44
4.2.4 Conditional proofs . . . 45
4.3 Automated random testing . . . 46
4.4 Checking satisfiability . . . 47

5 AES: The Advanced Encryption Standard **49**
5.1 Parameters . . . 49
5.2 Polynomials in GF(2^8) . . . 50
5.3 The `SubBytes` transformation . . . 51
5.4 The `ShiftRows` transformation . . . 53
5.5 The `MixColumns` transformation . . . 54
5.6 Key expansion . . . 55
5.7 The `AddRoundKey` transformation . . . 57
5.8 AES encryption . . . 57
5.9 Decryption . . . 58
5.9.1 The `InvSubBytes` transformation . . . 58
5.9.2 The `InvShiftRows` transformation . . . 59
5.9.3 The `InvMixColumns` transformation . . . 59
5.10 The inverse cipher . . . 60
5.11 Correctness . . . 60

A Solutions to selected exercises **63**
1.2 Bits: Booleans . . . 63
1.3 Words: Numbers . . . 63
1.4 Integers: Unbounded numbers . . . 63
1.7 Tuples: Heterogeneous collections . . . 64
1.8 Sequences: Homogeneous collections . . . 64
1.9 Words as sequences . . . 68
1.11 Records: Named collections . . . 69

1.12 The `zero` . . . 70
1.13 Arithmetic . . . 70
1.14 Types . . . 71
1.15 Defining functions . . . 72
1.16 Recursion and recurrences . . . 74
1.17 Stream equations . . . 76
1.18 Type synonyms . . . 76
2.1 Caesar's cipher . . . 76
2.2 Vigenère cipher . . . 78
2.3 The atbash . . . 79
2.4 Substitution ciphers . . . 79
2.5 The scytale . . . 80
3.1 The plugboard . . . 81
3.2 Scrambler rotors . . . 81
3.3 Connecting the rotors: notches in action . . . 81
3.4 The reflector . . . 83
3.5 Putting the pieces together . . . 83
3.7 Encryption and decryption . . . 84
4.1 Writing properties . . . 84
4.2 Establishing correctness . . . 85
4.3 Automated random testing . . . 87
4.4 Checking satisfiability . . . 88
5.2 Polynomials in GF(2^8) . . . 88
5.3 The `SubBytes` transformation . . . 90
5.4 The `ShiftRows` transformation . . . 91
5.5 The `MixColumns` transformation . . . 91
5.6 Key expansion . . . 92
5.8 AES encryption . . . 92
5.9 Decryption . . . 93

B Cryptol prelude functions **95**

C Enigma simulator **99**

D AES in Cryptol **103**

E Technicalities **107**
E.1 Language features . . . 107
E.2 Commands . . . 108

F Cryptol Syntax **111**
F.1 Layout . . . 111
F.2 Comments . . . 111
F.3 Identifiers . . . 111
F.4 Keywords and Built-in Operators . . . 112
F.5 Built-in Type-level Operators . . . 112
F.6 Numeric Literals . . . 113
F.7 Expressions . . . 114
F.8 Bits . . . 115
F.9 Multi-way Conditionals . . . 115
F.10 Tuples and Records . . . 115
F.10.1 Accessing Fields . . . 115
F.10.2 Updating Fields . . . 116
F.11 Sequences . . . 117
F.12 Functions . . . 117

F.13 Local Declarations 117
F.14 Explicit Type Instantiation 118
F.15 Demoting Numeric Types to Values 118
F.16 Explicit Type Annotations 118
F.17 Type Signatures 118
F.18 Type Synonym Declarations 118
F.19 Modules 118
F.20 Hierarchical Module Names 119
F.21 Module Imports 119
F.22 Import Lists 119
F.23 Hiding Imports 120
F.24 Qualified Module Imports 120
F.25 Private Blocks 120
F.26 Parameterized Modules 121
F.27 Named Module Instantiations 121
F.28 Parameterized Instantiations 122
F.29 Importing Parameterized Modules 123

Glossary **125**

Bibliography **127**

Index **129**

Chapter 1

A Crash Course in Cryptol

Before we can delve into cryptography, we have to get familiar with Cryptol. This chapter provides an introduction to Cryptol, just to get you started. The exposition is not meant to be comprehensive, but rather as an overview to give you a feel of the most important tools available. If a particular topic appears hard to approach, feel free to skim it over for future reference.

A full language reference is beyond the scope of this document at this time.

1.1 Basic data types

Cryptol provides seven basic data types: bits, sequences, integers, integers-modulo-n, rationals, tuples, and records. Words (i.e., n-bit numbers) are a special case of sequences. Note that, aside from the base types (like bits and integers), Cryptol types can be nested as deep as you like. That is, we can have records of sequences containing tuples made up of other records, etc., giving us a rich type-system for precisely describing the shapes of data our programs manipulate.

While Cryptol is statically typed, it uses type inference to supply unspecified types. That is, the user usually does *not* have to write the types of all expressions; they will be automatically inferred by the type-inference engine. Of course, in certain contexts the user might choose to supply a type explicitly. The notation is simple: we simply put the expression, followed by `:` and the type. For instance,

```
12 : [8]
```

means the value `12` has type `[8]`, i.e., it is an 8-bit word. We can also supply partial types. For example,

```
12 : [_]
```

means that `12` has a word type with an unspecified number of bits. Cryptol will infer the unspecified part of the type in this case. We shall see many other examples of typed expressions in the following discussion.

1.2 Bits: Booleans

The type `Bit` represents a single bit of information. There are precisely two values of this type: `True` and `False`. Bit values play an important role in Cryptol, as we shall see in detail shortly. In particular, the test expression in an `if-then-else` statement must have the type `Bit`. The logical operators `&&` (and), `||` (or), `^` (xor), and `~` (complement) provide the basic operators that act on bit values.

Exercise 1.1. Type in the following expressions at the Cryptol prompt, and observe the output:

```
True
false
False : Bit
if True && False then 0x3 else 0x4
False || True
```

```
(True && False) ^ True
~False
~(False || True)
```

Remember that Cryptol is case sensitive, and hence `false` is different from `False`.

Tip: Cryptol provides extensive command line/tab completion; use up/down-arrow to see your previous commands, hit tab to complete identifier names, etc.

1.3 Words: Numbers

A word is simply a numeric value with some number of bits, corresponding to the usual notion of binary numbers. To match our observation of how cryptographers use binary numbers, Cryptol defaults to interpreting words as non-negative (≥ 0) values (i.e., no negative numbers). However, words can be arbitrarily large: There is no predefined maximum number of bits that we are limited to. The type of n-bit words is written `[n]`; e.g. `[8]` is the 8-bit word type.

By default, Cryptol prints words in base 16. You might find it useful to set the output base to be 10 while working on the following example. To do so, use the command:

```
:set base=10
```

The supported values for this setting are 2 (binary), 8 (octal), 10 (decimal), and 16 (hexadecimal). Conversely, we can *write* numbers in these bases in Cryptol programs too:

```
0b11111011110     // binary
0o3736            // octal
2014              // decimal
0x7de             // hexadecimal
```

Decimal numbers pose a problem in a bit-precise language like Cryptol. Numbers represented in a base that is a power of two unambiguously specify the number of bits required to store each digit. For example `0b101` takes three bits to store, so its type is `[3]`. A hexadecimal digit takes 4 bits to store, so `0xabc` needs 12 bits, and its type is `[12]`. On the other hand, a decimal digit could require anywhere from 1 to 4 bits to represent, so the number of digits does not determine the type. Decimal numbers may assume any of a variety of types in Cryptol; that is, they are *polymorphic*. For example, `19` could have the unbounded type `Integer` (see section 1.4), type `[8]`, type `[5]`, or any other word type with at least 5 bits.

Defaulting and explicit types Polymorphic values in a Cryptol program are subject to *defaulting*, meaning that Cryptol will choose specific types to represent them. Cryptol usually prints a message whenever defaulting occurs, showing which type was chosen. For numbers, Cryptol usually chooses either `Integer` or the word type with the *smallest* number of bits required to represent them. Users can override this by giving an explicit type signature.

Exercise 1.2. Try writing some decimal numbers at different types, like `12:[4]` and `50:[8]`. How does Cryptol respond if you write `19:[4]`, and why? What is the largest decimal number you can write at type `[8]`? What is the smallest allowable bit size for `32`? What is the smallest allowable bit size for `0`?

Exercise 1.3. Experiment with different output bases by issuing `:set base=10`, and other base values. Also try writing numbers directly in different bases at the command line, such as `0o1237`. Feel free to try other bases. What is the hexadecimal version of the octal number `0o7756677263570015`?

Note: We will revisit the notion of words in section 1.9, after we learn about sequences.

1.4 Integers: Unbounded numbers

The type `Integer` represents mathematical integers of unlimited size. To write an integer value in Cryptol, simply write the number in decimal, optionally annotated with `:Integer` to disambiguate the type. Numbers written in base 2, 8, or 16 are always *n*-bit words, and never have type `Integer`. However, they can be converted to unbounded integers using `toInteger`. For example, `toInteger 0xff` represents the integer 255. To write a negative integer, simply negate the result: For example, `-12` and `- toInteger 0xc` represent the integer −12.

Exercise 1.4. Compare the results of evaluating the expressions `0x8 + 0x9` and `toInteger 0x8 + toInteger 0x9`. Also try evaluating `toInteger (- 0x5)`. Can you explain the results?

In addition to the unbounded `Integer` type, there are also a collection of bounded integer types: `Z n` for each finite positive `n`. `Z n` represents the ring of integers modulo `n`. Just like with fixed-width bitvector values, arithmetic in `Z n` will "wrap around" (i.e., be reduced modulo `n`) when values get too large.

Exercise 1.5. Compare the results of evaluating `(5 : Z 7) + (6 : Z 7)` with `toInteger (5 : Z 7) + toInteger (6 : Z 7)`. What is the result of `-3 : Z 7`? Were the results what you expected?

1.5 Rationals

The type `Rational` represents the rational number subset of the real line: that is, those numbers that can be represented as a ratio of integers. You can explicitly create rational values using the `ratio` function, which accepts the numerator and denominator as integer values, or you can create a rational value from an integer using the `fromInteger` function. Rational values can also be created directly from literals the same way as for `Integer`.

For example, all of the following expression create the same value representing one half:

```
ratio 1 2
ratio 2 4
(1 /. 2) : Rational
(recip 2) : Rational
(fromInteger (-1) /. fromInteger (-2)) : Rational
```

Note, the division operation on `Rational` is written `/.`, and should not be confused with `/`, which is the division operation for words and integers.

1.6 Floating Point Numbers

The family of types `Float e p` represent IEEE 754 floating point numbers with `e` bits in the exponent and `p-1` bits in the mantissa. The family is defined in a built-in module called `Float` so to use it you'd need to either import it in you Cryptol specification or use `:m Float` on the Cryptol REPL.

Floating point numbers may be written using either integral or fractioanl literals. In general, literals that cannot be represented exactly are rejected with a type error, with the exception of decimal fractional litrals which are rounded to the nearest even representable numbers.

Floating point numbers may be manipulated and compared using standard Cryptol operators, such as `+`, `==`, and `/.` (but *not* `/` which is only for integral types). When using the standard Cryptol operators the results that cannot be represented exactly are rounded to the nearest even representable number.

The module `Float` provides additional functionality specific to floating point numbers. In particular, it contains constants for writing special floating point values, and arithmetic operations that are parameterized on the rounding mode to use during computation (e.g., `fpAdd`).

Of particular importance is the relation `=.=` for comparing floating point numbers semantically, rather that using the IEEE equality, which is done by `==`. The behavior of the two differs mostly on special values, here are some examples:

```
Float> fpNaN == (fpNaN : Float64)
False
Float> fpNaN =.= (fpNaN : Float64)
True
Float> 0 == (-0 : Float64)
True
Float> 0 =.= (-0 : Float64)
False
```

Cryptol supports reasoning about floating point numbers using the `What4` interface to solvers that support IEEE floats. These back-ends have names starting with `w4`, for example, a good one to try is `w4-z3`.

The Cryptol REPL has some settings to control how floating point numbers are printed on the REPL. In particular `:fp-base` specifies in what numeric base to output floating point numbers, while `:fp-format` provides various settings for formatting the results. For more information, see `:help :set fp-format`.

1.7 Tuples: Heterogeneous collections

A tuple is a simple ordered collection of arbitrary values with arbitrary types. Tuples are written with parentheses surrounding two or more comma-separated elements. Cryptol also supports a special zero-element tuple, written `()`.[1]

Tuple types have syntax similar to tuple values. They are written with parentheses around two or more comma-separated types. Values of a tuple type are tuples of the same length, where the value in each position has the corresponding type. For example, type `([8], Bit)` comprises pairs whose first element is an 8-bit word, and whose second element is a bit. The empty tuple `()` is the only value of type `()`.

Exercise 1.6. Try out the following tuples:

```
(1, 2+4)
(True, False, True ^ False)
((1, 2), False, (3-1, (4, True)))
```

Projecting values from tuples Use a `.` followed by n to project the $n+1$-th component of a tuple. Nested projections are also supported. Note that projections cannot be used with the empty tuple, as it has no components to project.

Exercise 1.7. Try out the following examples:

```
(1, 2+4).0
(1, 2+4).1
((1, 2), False, (3-1, (4, True))).2.1
```

Write a nested projection to extract the value `True` from the expression:

```
((1, 2), (2, 6, (True, 4)), False)
```

Tip: While tuple projections can come in handy, we rarely see them used in Cryptol programs. As we shall see later, Cryptol's powerful pattern-matching mechanism provides a much nicer and usable alternative for extracting parts of tuples and other composite data values.

1.8 Sequences: Homogeneous collections

While tuples contain heterogeneous data, sequences are used for homogeneous collections of values, akin to value arrays in more traditional languages. A sequence contains elements of any *single* type, even sequences themselves, arbitrarily nested. We simply write a sequence by enclosing it within square brackets with comma-separated elements.

Exercise 1.8. Try out the following sequences:

```
[1, 2]
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Note how the latter example can be used as the representation of a 3×3 matrix.

Tip: The most important thing to remember about a sequence is that its elements must be of exactly the same type.

Exercise 1.9. Type in the following expressions to Cryptol and observe the type errors:

[1] There is no syntax for tuples with exactly one element. Writing parentheses around a single value does not make a tuple; it denotes the same value with the same type.


```
[True, [True]]
[[1, 2, 3], [4, 5]]
```

1.8.1 Enumerations

Cryptol enumerations allow us to write sequences more compactly, instead of listing the elements individually. An enumeration is a means of writing a sequence by providing a (possibly infinite) range. Cryptol enumerations are not equivalent to mainstream programming languages' notions of enumeration types, other than both kinds of constructs guarantee that enumeration elements are distinct.

Exercise 1.10. Explore various ways of constructing enumerations in Cryptol, by using the following expressions:

```
[1 .. 10]             // increment with step 1
[1, 3 .. 10]          // increment with step 2 (= 3-1)
[10, 9 .. 1]          // decrement with step 1 (= 10-9)
[10, 9 .. 20]         // decrement with step 1 (= 10-9)
[10, 7 .. 1]          // decrement with step 3 (= 10-7)
[10, 11 .. 1]         // increment with step 1
[1 .. 10 : [8]]       // increment 8-bit words with step 1
[1, 3 .. 10 : [16]]   // increment 16-bit words with step 2 (= 3-1)
```

1.8.2 Comprehensions

A Cryptol comprehension is a way of programmatically computing the elements of a new sequence, out of the elements of existing ones. The syntax is reminiscent of the set comprehension notation from ordinary mathematics, generalized to cover parallel branches (as explained in the exercises below). Note that Cryptol comprehensions are not generalized numeric comprehensions (like summation, product, maximum, or minimum), though such comprehensions can certainly be defined using Cryptol comprehensions.

Exercise 1.11. The components of a Cryptol sequence comprehension are an expression of one or more variables (which defines each element of the sequence), followed by one or more *branches*, each preceded by a vertical bar, which define how the variables' values are generated. A comprehension with a single branch is called a *cartesian comprehension*. We can have one or more components in a cartesian comprehension. Experiment with the following expressions:

```
[ (x, y) | x <- [1 .. 3], y <- [4, 5] ]
[ x + y  | x <- [1 .. 3], y <- [] ]
[ (x + y, z) | x <- [1, 2], y <- [1], z <- [3, 4] ]
```

What is the number of elements in the resulting sequence, with respect to the sizes of components?

Exercise 1.12. A comprehension with multiple branches is called a *parallel comprehension*. We can have any number of parallel branches. The contents of each branch will be *zipped* to obtain the results. Experiment with the following expressions:

```
[ (x, y) | x <- [1 .. 3] | y <- [4, 5] ]
[ x + y  | x <- [1 .. 3] | y <- [] ]
[ (x + y, z)  | x <- [1, 2] | y <- [1] | z <- [3, 4] ]
```

What is the number of elements in the resulting sequence, with respect to the sizes of the parallel branches?

Tip: One can mix parallel and cartesian comprehensions, where each parallel branch can contain multiple cartesian generators.

Tip: While Cryptol comprehensions *look* like standard mathematical comprehensions, one must remember that the codomain of Cryptol comprehensions is a sequence type of some kind, *not* a set.

Comprehensions may be nested. In this pattern, the element value expression of the outer nesting is a sequence comprehension (which may refer to values generated by the outer generator). The pattern looks like this:

```
[  [ <expr with x & y>  //                    \
     | y <- [1 .. 5]    // inner generator     -- outer
   ]                                          /  elements
   | x <- [1 .. 5]      // outer generator
]
```

Exercise 1.13. Use a nested comprehension to write an expression to produce a 3×3 matrix (as a sequence of sequences), such that the *ij*th entry contains the value `(i, j)`.

1.8.3 Appending and indexing

For sequences, the two basic operations are appending (`#`) and selecting elements out (`@`, `@@`, `!`, and `!!`). The forward selection operator (`@`) starts counting from the beginning, while the backward selection operator (`!`) starts from the end. Indexing always starts at zero: that is, `xs @ 0` is the first element of `xs`, while `xs ! 0` is the last. The permutation versions (`@@` and `!!`, respectively) allow us to concisely select multiple elements: they allow us to extract elements in any order (which makes them very useful for permuting sequences).

Exercise 1.14. Try out the following Cryptol expressions:

```
[] # [1, 2]
[1, 2] # []
[1 .. 5] # [3, 6, 8]
[0 .. 9] @ 0
[0 .. 9] @ 5
[0 .. 9] @ 10
[0 .. 9] @@ [3, 4]
[0 .. 9] @@ []
[0 .. 9] @@ [9, 12]
[0 .. 9] @@ [9, 8 .. 0]
[0 .. 9] ! 0
[0 .. 9] ! 3
[0 .. 9] !! [3, 6]
[0 .. 9] !! [0 .. 9]
[0 .. 9] ! 12
```

Exercise 1.15. The permutation operators (`@@` and `!!`) can be defined using sequence comprehensions. Write an expression that selects the even indexed elements out of the sequence `[0 .. 10]` first using `@@`, and then using a sequence comprehension.

1.8.4 Finite and infinite sequences

So far we have only seen finite sequences. An infinite sequence is one that has an infinite number of elements, corresponding to streams. Cryptol supports infinite sequences, where the elements are accessed *on demand*. This implies that Cryptol will *not* go into an infinite loop just because you have created an infinite sequence: it will lazily construct the sequence and make its elements available as demanded by the program.

Exercise 1.16. Try the following infinite enumerations:

```
[1:[32] ...]
[1:[32], 3 ...]
[1:[32] ...] @ 2000
[1:[32], 3 ...] @@ [300, 500, 700]
[100, 102 ...]
```

Note: We are explicitly telling Cryptol to use 32-bit words as the elements. The reason for doing so will become clear when we study arithmetic shortly.

Exercise 1.17. What happens if you use the reverse index operator (`!`) on an infinite sequence? Why?

1.8.5 Manipulating sequences

Sequences are at the heart of Cryptol, and there are a number of built-in functions for manipulating them in various ways. It is worthwhile to try the following exercises to gain basic familiarity with the basic operations.

Exercise 1.18. Try the following expressions:

```
take`{3} [1 .. 12]
drop`{3} [1 .. 12]
split`{3} [1 .. 12]
groupBy`{3} [1 .. 12]
join [[1 .. 4], [5 .. 8], [9 .. 12]]
join [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
transpose [[1, 2, 3, 4], [5, 6, 7, 8]]
transpose [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

And for fun, think about what these should produce:

```
join [1,1]
transpose [1,2]
```

Exercise 1.19. Based on your intuitions from the previous exercise, derive laws between the following pairs of functions: `take` and `drop`; `join` and `split`; `join` and `groupBy`; `split` and `groupBy` and `transpose` and itself. For instance, `take` and `drop` satisfy the following equality:

```
(take`{n} xs) # (drop`{n} xs) == xs
```

whenever `n` is between `0` and the length of the sequence `xs`. Note that there might be multiple laws these functions satisfy.

Exercise 1.20. What is the relationship between the append operator `#` and `join`?

Type-directed splits The Cryptol primitive function `split` splits a sequence into any number of equal-length parts. An explicit result type is often used with `split`, since the number of parts and the number of elements in each part are not given as arguments, but are determined by the type of the argument sequence and the result context.

```
Cryptol> split [1..12] : [1][12][8]
[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]]
Cryptol> split [1..12] : [2][6][8]
[[1, 2, 3, 4, 5, 6], [7, 8, 9, 10, 11, 12]]
Cryptol> split [1..12] : [3][4][8]
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

Here is what happens if we do *not* give an explicit signature on the result:

```
Cryptol> split [1..12]
Cannot evaluate polymorphic value.
Type: {n, m, a} (n * m == 12, Literal 12 a, fin m) => [n][m]a
Cryptol> :t split [1..12]
split [1 .. 12] : {n, m, a} (n * m == 12, Literal 12 a, fin m) =>
                    [n][m]a
```

A complex type signature like this one first defines a set of type variables `{n, m, a}`, a set of constraints on those variables (`n * m == 12, Literal 12 a, fin m`), a `=>` and finally the shape description. In this case, Cryptol's `[n][m]a` is telling us that the result will be a sequence of `n` things, each of which is a sequence of `m` things, each of which is a value of type `a`. The `Literal` constraint tells us that `a` is a type that can represent numbers (at least) up to

12. The other constraints are that `n * m == 12`, which means we should completely cover the entire input, and that the length `m` needs to be finite. As you can see, `split` is a very powerful function. The flexibility afforded by `split` comes in very handy in Cryptol. We shall see one example of its usage later in section 2.5.

Exercise 1.21. With a sequence of length 12, as in the above example, there are precisely 6 ways of splitting it: 1×12, 2×6, 3×4, 4×3, 6×2, and 12×1. We have seen the first three splits above. Write the expressions corresponding to the latter three.

Exercise 1.22. What happens when you type `split [1 .. 12] : [5][2][8]`?

Exercise 1.23. Write a `split` expression to turn the sequence `[1 .. 120] : [120][8]` into a nested sequence with type `[3][4][10][8]`, keeping the elements in the same order. (**Hint** Use nested comprehensions.)

1.8.6 Shifts and rotates

Common operations on sequences include shifting and rotating them. Cryptol supports both versions with left/right variants.

Exercise 1.24. Experiment with the following expressions:

```
[1, 2, 3, 4, 5] >> 2
[1, 2, 3, 4, 5] >> 10
[1, 2, 3, 4, 5] << 2
[1, 2, 3, 4, 5] << 10
[1, 2, 3, 4, 5] >>> 2
[1, 2, 3, 4, 5] >>> 10
[1, 2, 3, 4, 5] <<< 2
[1, 2, 3, 4, 5] <<< 10
```

Notice that shifting/rotating always returns a sequence precisely the same size as the original.

Exercise 1.25. Let `xs` be a sequence of length n. What is the result of rotating `xs` left or right by a multiple of n?

1.9 Words as sequences

In section 1.3 we have introduced numbers as a distinct value type in Cryptol. In fact, a number in Cryptol is nothing but a finite sequence of bits, so words are not a separate type. For instance, the literal expression `42:[6]` is precisely the same as the bit-sequence `[True, False, True, False, True, False]`. The type `[6]` is really just an abbreviation for `[6]Bit`.

Exercise 1.26. Explain why 42 is the same as `[True, False, True, False, True, False]`. Is Cryptol little-endian, or big-endian?

Exercise 1.27. Try out the following words: (**Hint** It might help to use `:set base=2` to see the bit patterns.)

```
12:[4]
12 # [False]
[False, False] # 12
[True, False] # 12
12 # [False, True]
32:[6]
12 # 32
[True, False, True, False, True, False] == 42
```

Exercise 1.28. Since words are sequences, the sequence functions from exercise 1.18 apply to words as well. Try out the following examples and explain the outputs you observe:


```
take`{3} 0xFF
take`{3} (12:[6])
drop`{3} (12:[6])
split`{3} (12:[6])
groupBy`{3} (12:[6])
```

Recall that the notation `12:[6]` means the constant 12 with the type precisely 6 bits wide.

Exercise 1.29. Try exercise 1.28, this time with the constant `12:[12]`. Do any of the results change? Why?

Shifts and rotates on words Consider what happens if we shift a word, say `12:[6]` by one to the right:

```
(12:[6]) >> 1
  = [False, False, True, True, False, False] >> 1
  = [False, False, False, True, True, False]
  = 6
```

That is shifting right by one effectively divides the word by 2. This is due to Cryptol's "big-endian" representation of numbers.[3]

Exercise 1.30. Try the following examples of shifting/rotating words:

```
(12:[8]) >> 2
(12:[8]) << 2
```

Little-endian vs Big-endian The discussion of endianness comes up often in computer science, with no clear winner. Since Cryptol allows indexing from the beginning or the end of a (finite) sequence, you can access the 0th (least-significant) bit of a sequence `k` with `k!0`, the 1st bit with `k!1`, and so on.

1.10 Characters and strings

Strictly speaking Cryptol does *not* have characters and strings as a separate type. However, Cryptol does allow characters in programs, which simply correspond to their ASCII equivalents. Similarly, strings are merely sequences of characters, i.e., sequences of words. The following examples illustrate:

```
Cryptol> :set base=10
Cryptol> :set ascii=off
Cryptol> 'A'
65
Cryptol> "ABC"
[65, 66, 67]
Cryptol> :set ascii=on
Cryptol> "ABC"
"ABC"
Cryptol> :set ascii=off
Cryptol> ['A' .. 'Z']
[65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
     80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90]
Cryptol> :set ascii=on
Cryptol> ['A' .. 'Z']
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

[3]This is a significant change from Cryptol version 1, which interpreted the leftmost element of a sequence as the lowest-ordered bit (and thus shifting right was multiplying by 2, and shifting left was dividing by 2). The way it is handled now matches the traditional interpretation.

Note: This is the reason why we have to use the `:set ascii=on` command to print ASCII strings. Otherwise, Cryptol will not have enough information to tell numbers from characters.
Since characters are simply 8-bit words, you can do word operations on them; including arithmetic:

```
Cryptol> 'C' - 'A'
2
```

1.11 Records: Named collections

In Cryptol, records are simply collections of named fields. In this sense, they are very similar to tuples (section 1.7), which can be thought of records without field names. Like a tuple, the fields of a record can be of any type. We construct records by listing the fields inside curly braces, separated by commas. We project fields out of a record with the usual dot-notation, similar to tuple projections. Note that the order of fields in a record is immaterial.

Record equality is defined in the standard fashion. Two records are equal if they have the same number of fields, if their field names are identical, if identically named fields are of comparable types and have equal values.

Exercise 1.31. Type in the following expressions and observe the output:

```
{xCoord = 12:[32], yCoord = 21:[32]}
{xCoord = 12:[32], yCoord = 21:[32]}.yCoord
{name = "Cryptol", address = "Galois"}
{name = "Cryptol", address = "Galois"}.address
{name = "test", coords = {xCoord = 3:[32], yCoord = 5:[32]}}
{name = "test", coords = {xCoord = 3:[32], \
                          yCoord = 5:[32]}}.coords.yCoord
{x=True, y=False} == {y=False, x=True}
```

You might find the command `:set ascii=on` useful in viewing the output.

1.12 The `zero`

Before proceeding further, we have to take a detour and talk briefly about one of the most useful values in Cryptol: `zero`. The value `zero` inhabits every type in Cryptol. The following examples should illustrate the idea:

```
Cryptol> zero : Bit
False
Cryptol> zero : [8]
0
Cryptol> zero : Integer
0
Cryptol> zero : Z 19
0
Cryptol> zero : Rational
(ratio 0 1)
Cryptol> zero : ([8], Bit)
(0, False)
Cryptol> zero : [8][3]
[0, 0, 0, 0, 0, 0, 0, 0]
Cryptol> zero : [3](Bit, [4])
[(False, 0), (False, 0), (False, 0)]
Cryptol> zero : {xCoord : [12], yCoord : [5]}
{xCoord=0, yCoord=0}
```

On the other extreme, the value `zero` combined with the complement operator `~` gives us values that consist of all `True` bits:


```
Cryptol> ~zero : Bit
True
Cryptol> ~zero : [8]
255
Cryptol> ~zero : ([8], Bit)
(255, True)
Cryptol> ~zero : [8][3]
[7, 7, 7, 7, 7, 7, 7, 7]
Cryptol> ~zero : [3](Bit, [4])
[(True, 15), (True, 15), (True, 15)]
Cryptol> ~zero : {xCoord : [12], yCoord : [5]}
{xCoord=4095, yCoord=31}
```

Exercise 1.32. We said that `zero` inhabits all types in Cryptol. This also includes functions. What do you think the appropriate `zero` value for a function would be? Try out the following examples:

```
(zero : ([8] -> [3])) 5
(zero : Bit -> {xCoord : [12], yCoord : [5]}) True
```

1.13 Arithmetic

Cryptol supports the usual binary arithmetic operators `+`, `-`, `*`, `^^` (exponentiation), `/` (integer division), `%` (integer modulus), along with *ceiling* base-2 logarithm `lg2` and binary `min` and `max`.

It is important thing to remember is that all arithmetic in Cryptol is carried out according to the type of the values being computed; for word types in particular, arithmetic is modular, with respect to the underlying word size. As a consequence, there is no such thing as an overflow/underflow in Cryptol, as the result will be always guaranteed to fit in the resulting word size. While this is very handy for most applications of Cryptol, it requires some care if overflow has to be treated explicitly.

Exercise 1.33. What is the value of `1 + 1 : [_]`? Surprised?

Exercise 1.34. What is the value of `1 + 1 : [8]`? Why?

Exercise 1.35. What is the value of `3 - 5 : [_]`? How about `3 - 5 : [8]`?

Note: Cryptol supports subtraction both as a binary operator, and as a unary operator. When used in a unary fashion (a.k.a. unary minus), it simply means subtraction from `0`. For instance, `-5` precisely means `0-5`, and is subject to the usual modular arithmetic rules.

Exercise 1.36. Try out the following expressions:

```
2 / 0
2 % 0
3 + (if 3 == 2+1 then 12 else 2/0)
3 + (if 3 != 2+1 then 12 else 2/0)
lg2 (-25) : [_]
```

In the last expression, remember that unary minus will be done in a modular fashion. What is the modulus used for this operation?

Exercise 1.37. Integral division truncates down. Try out the following expressions:

```
(6 / 3, 6 % 3)
(7 / 3, 7 % 3)
(8 / 3, 8 % 3)
(9 / 3, 9 % 3)
```


What is the relationship between `/` and `%`?

Exercise 1.38. What is the value of `min (5:[_]) (-2)`? Why? Why are the parentheses around `-2` necessary?

Exercise 1.39. How about `max 5 (-2:[8])`? Why?

Exercise 1.40. Write an expression that computes the sum of two sequences `[1.. 10]` and `[10, 9 .. 1]`.

Comparison operators Cryptol supports the comparison operators `==`, `!=`, `>`, `>=`, `<`, `<=`, with their usual meanings.

Exercise 1.41. Try out the following expressions:

```
((2 >= 3) || (3 < 6)) && (4 == 5)
if 3 >= 2 then True else 1 < 12
```

Enumerations, revisited In exercise 1.16, we wrote the infinite enumeration starting at `1` using an explicit type as follows:

```
[(1:[32]) ...]
```

As expected, Cryptol evaluates this expression to:

```
[1, 2, 3, 4, 5, ...]
```

However, while the output suggests that the numbers are increasing all the time, that is just an illusion! Since the elements of this sequence are 32-bit words, eventually they will wrap over, and go back to 0. (In fact, this will happen precisely at the element $2^{32} - 1$, starting the count at 0 as usual.) We can observe this much more simply, by using a smaller bit size for the constant `1`:

```
Cryptol> [(1:[2])...]
[1, 2, 3, 0, 1 ...
Cryptol> take`{20} [(1:[2])...]
[1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0]
```

We still get an infinite sequence, but the numbers will repeat themselves eventually. Note that this is a direct consequence of Cryptol's modular arithmetic.

There is one more case to look at. What happens if we completely leave out the bit-width?

```
Cryptol> [1:[_] ...]
[1, 0, 1, 0, 1, ...]
```

In this case, Cryptol figured out that the number `1` requires precisely 1 bit, and hence the arithmetic is done modulo $2^1 = 2$, giving us the sequence $1, 0, 1, 0, \ldots$ In particular, an enumeration of the form:

```
[k ...]
```

will be treated as if the user has written:

```
[k, (k+1) ...]
```

and type inference will assign the smallest bit-size possible to represent `k`.

Exercise 1.42. Remember from exercise 1.2 that the word `0` requires 0 bits to represent. Based on this, what is the value of the enumeration `[0:[_], 1...]`? How about `[0:[_]...]`? Surprised?

Exercise 1.43. What is the value of `[1 .. 10]`? Explain in terms of the above discussion on modular arithmetic.

1.14 Types

Cryptol's type system is one of its key features.[4] You have seen that types can be used to specify the exact width of values, or shapes of sequences using a rich yet concise notation. In some cases, it may make sense to omit a type signature and let Cryptol *infer* the type for you. At the interpreter, you can check what type Cryptol inferred with the `:t` command.

1.14.1 Monomorphic types

A monomorphic type is one that represents a concrete value. Most of the examples we have seen so far fall into this category. Below, we review the basic Cryptol types that make up all the monomorphic values in Cryptol.

Bits There are precisely two bit values in Cryptol: `True` and `False`. The type itself is written `Bit`. When we want to be explicit, we can write it as follows: `(2 >= 3) : Bit`. However, with type inference writing the `Bit` type explicitly is almost never needed.

Words A word type is written `[`n`]`, where n is a fixed non-negative constant. The constant can be as large (or small) as you like. So, you can talk about 2-bit quantities `[2]`, as well as 384-bit ones `[384]`, or even odd sizes like 17 `[17]`, depending on the needs of your application. When we want to be explicit about the type of a value, we say `5:[8]`. If we do not specify a size, Cryptol's type inference engine will pick the appropriate value depending on the context. Recall from section 1.9 that a word is, in fact, a sequence of bits. Hence, an equivalent (but verbose) way to write the type `[17]` is `[17]Bit`, which we would say in English as "a sequence of length 17, whose elements are Bits."

Integers The type `Integer` represents the unbounded mathematical integers. Arithmetic on this type will not overflow.

Modular integers The type `Z n` represents the subset of the integers from 0 to `n` (not including `n`). Arithmetic on these types is done modulo `n`.

Rationals The type `Rational` represents the subset of the real line that can be represented as a ratio of integers. As with the integers, arithmetic on this type does not overflow.

Tuples A tuple is a heterogeneous collection of arbitrary number of elements. Just like we write a tuple value by enclosing it in parentheses, we write the tuple type by enclosing the component types in parentheses, separated by commas: `(3, 5, True) : ([8], [32], Bit)`. Tuples' types follow the same structure: `(2, (False, 3), 5) : ([8], (Bit, [32]), [32])`. A tuple component can be any type: a word, another tuple, sequence, record, etc. Again, type inference makes writing tuple types hardly ever necessary.

Sequences A sequence is simply a collection of homogeneous elements. If the element type is t, then we write the type of a sequence of n elements as `[`n`]`t. Note that t can be a sequence type itself. For instance, the type `[12][3][6]` reads as follows: A sequence of 12 elements, each of which is a sequence of 3 elements, each of which is a 6-bit-wide word.

The type of an infinite sequence is written `[inf]`t, where t is the type of the elements.

Exercise 1.44. What is the total number of bits in the type `[12][3][6]`?

Exercise 1.45. How would you write the type of an infinite sequence where each element itself is an infinite sequence of 32-bit words? What is the total bit size of this type?

Records A record is a heterogeneous collection of arbitrary number of labeled elements. In a sense, they generalize tuples by allowing the programmer to give explicit names to fields. The type of a record is written by enclosing it in braces, separated by commas: `{x : [32], y : [32]}`. Records can be nested and can contain arbitrary types of elements (records, sequences, functions, etc.).

[4] The Cryptol type system is based on the traditional Hindley-Milner style, extended with size types and arithmetic predicates (for details, see [5, 6, 7])

1.14.2 Polymorphic types

Our focus so far has been on monomorphic types—the types that concrete Cryptol values (such as `True`, `3`, or `[1, 2]`) can have. If all we had were monomorphic types, however, Cryptol would be a very verbose and boring language. Instead, we would like to be able to talk about collections of values, values whose types are instances of a given polymorphic type. This facility is especially important when we define functions, a topic we will get to shortly. In the mean time, we will look at some of the polymorphic primitive functions Cryptol provides to get a feeling for Cryptol's polymorphic type system.

The tale of `tail` Cryptol's built in function `tail` allows us to drop the first element from a sequence, returning the remainder:

```
Cryptol> tail [1 .. 5]
[2, 3, 4, 5]
Cryptol> tail [(False, (1:[8])), (True, 12), (False, 3)]
[(True, 12), (False, 3)]
Cryptol> tail [ (1:[16])... ]
[2, 3, 4, 5, 6, ...]
```

What exactly is the type of `tail`? If we look at the first example, one can deduce that `tail` must have the type:

```
tail : [5][8] -> [4][8]
```

That is, it takes a sequence of length 5, containing 8-bit values, and returns a sequence that has length 4, containing 8-bit values. (The type `a -> b` denotes a function that takes a value of type `a` and delivers a value of type `b`.)

However, the other example uses of `tail` above suggest that it must have the following types, respectively:

```
tail : [10][32] -> [9][32]
tail : [3](Bit, [8]) -> [2](Bit, [8])
tail : [inf][16] -> [inf][16]
```

As we have emphasized before, Cryptol is strongly typed, meaning that every entity (whether a Cryptol primitive or a user-defined function) must have a well-defined type. Clearly, the types we provided for `tail` above are quite different from each other. In particular, the first example uses numbers as the element type, while the second has tuples. So, how can `tail` be assigned a type that will make it work on all these inputs?

If you are familiar C++ templates or Java generics, you might think that Cryptol has some sort of an overloading mechanism that allows one to define functions that can work on multiple types. While templates and generics do provide a mental model, the correspondence is not very strong. In particular, we never write multiple definitions for the same function in Cryptol, i.e., there is no ad-hoc overloading. However, what Cryptol has is a much stronger notion: polymorphism, as would be advocated by languages such as Haskell or ML [11, 13].

Here is the type of `tail` in Cryptol:

```
Cryptol> :t tail
tail : {n, a} [1 + n]a -> [n]a
```

This is quite a different type from what we have seen so far. In particular, it is a polymorphic type, one that can work over multiple concrete instantiations of it. Here's how we read this type in Cryptol:

> `tail` is a polymorphic function, parameterized over `n` and `a`. The input is a sequence that contains `1 + n` elements. The elements can be of an arbitrary type `a`; there is no restriction on their structure. The result is a sequence that contains `n` elements, where the elements themselves have the same type as those of the input.

In the case for `tail`, the parameter `n` is a size-parameter (since it describes the size of a sequence), while `a` is a shape-parameter, since it describes the shape of elements. The important thing to remember is that each use of `tail` must instantiate the parameters `n` and `a` appropriately. Let's see how the instantiations work for our running examples:

[n+1]a -> [n]a	n	a	Notes
[5][8] -> [4][8]	4	[8]	1+n = 5 ⇒ n = 4
[10][32] -> [9][32]	9	[32]	1+n = 10 ⇒ n = 9
[3](Bit, [8]) -> [2](Bit, [8])	2	(Bit, [8])	The type a is now a tuple
[inf][16] -> [inf][16]	inf	[16]	1+n = inf ⇒ n = inf

In the last instantiation, Cryptol knows that $\infty - 1 = \infty$, allowing us to apply `tail` on both finite and infinite sequences. The crucial point is that an instantiation must be found that satisfies the required match. It is informative to see what happens if we apply `tail` to an argument where an appropriate instantiation can not be found:

```
Cryptol> tail True
[error] at <interactive>:1:6--1:10:
  Type mismatch:
    Expected type: [1 + ?n`859]?a`860
    Inferred type: Bit
  where
  ?n`859 is type argument 'n' of 'tail' at <interactive>:1:1--1:5
  ?a`860 is type argument 'a' of 'tail' at <interactive>:1:1--1:5
```

Cryptol is telling us that it cannot match the types `Bit` and the sequence `[1+n]a`, causing a type error statically at compile time. (The funny notation of `?n`859` and `?a`860` are due to how type instantiations proceed under the hood. While they look funny at first, you soon get used to the notation.)

We should emphasize that Cryptol polymorphism uniformly applies to user-defined functions as well, as we shall see in section 1.15.

Exercise 1.46. Use the `:t` command to find the type of `split`. For each use case below, find out what the instantiations of its type variables are, and justify why the instantiation works. Can you find an instantiation in all these cases?

```
split`{3} [1..9]
split`{3} [1..12]
split`{3} [1..10] : [3][2][8]
split`{3} [1..10]
```

Is there any way to make the last example work by giving a type signature?

1.14.3 Predicates

In the previous section we have seen how polymorphism is a powerful tool in structuring programs. Cryptol takes the idea of polymorphism on sizes one step further by allowing predicates on sizes. To illustrate the notion, consider the type of the Cryptol primitive `take`:

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

The type of `take` says that it is parameterized over `front` and `back`, `front` must be a finite value, it takes a sequence `front + back` long, and returns a sequence `front` long.

The impact of this predicate shows up when we try to take more than what is available:

```
Cryptol> take`{10} [1..5]
[error] at <interactive>:1:11--1:17:
  Adding 10 will always exceed 5
```

Cryptol is telling us that it is unable to satisfy this instantiation (since `front` is 10 and the sequence has 5 elements).

In general, type predicates exclusively describe *arithmetic constraints on type variables*. Cryptol does not have a general-purpose dependent type system, but a *size-polymorphic type system*. Often type variables' values are of finite size, indicated with the constraint `fin a`; otherwise no constraint is mentioned, and the variables' values are unbounded, possibly taking the value `inf`. Arithmetic relations are arbitrary relations over all type variables, such as `2*a+b >= c`. We shall see more examples as we work through programs later on.

Exercise 1.47. Write a predicate that allows a word of size `128`, `192`, or `256`, but nothing else.

Note: Type inference in the presence of arithmetic predicates is an undecidable problem [6]. This implies that there is no algorithm to decide whether a given type is inhabited, amongst other things. In practical terms, we might end up writing programs with arbitrarily complicated predicates (e.g., this "type contains all solutions to Fermat's last equation" or "this type contains all primes between two large numbers") without Cryptol being able to simplify them, or notice that there is no instantiation that will ever work. Here is a simple example of such a type:

```
{k} (2 >= k, k >= 5) => [k]
```

While a moment of pondering suffices to conclude that there is no such value in this particular case, there is no algorithm to decide this problem in general.

That being said, Cryptol's type inference and type checking algorithms are well-tuned to the use cases witnessed in the types necessary for cryptographic algorithms. Moreover, Cryptol uses a powerful SMT solver capable of reasoning about complex arithmetic theories within these algorithms.

1.14.4 Why typed?

There is a spectrum of type systems employed by programming languages, all the way from completely untyped to fancier dependently typed languages. There is no simple answer to the question, what type system is the best? It depends on the application domain. We have found that Cryptol's size-polymorphic type system is a good fit for programming problems that arise in the domain of cryptography. The bit-precise type system makes sure that we never pass an argument that is 32 bits wide in a buffer that can only fit 16. The motto is: *Well typed programs do not go wrong.*

In practical terms, this means that the type system catches most of the common mistakes that programmers tend to make. Size-polymorphism is a good fit for Cryptol, as it keeps track of the most important invariant in our application domain: making sure the sizes of data can be very precisely specified and the programs can be statically guaranteed to respect these sizes.

Opponents of type systems typically argue that the type system gets in the way.[5] It is true that the type system will reject some programs that makes perfect sense. But what is more important is that the type system will reject programs that will indeed go wrong at run-time. And the price you pay to make sure your program type-checks is negligible, and the savings due to type checking can be enormous.

The crucial question is not whether we want type systems, but rather what type system is the best for a given particular application domain. We have found that the size-polymorphic type system of Cryptol provides the right balance for the domain of cryptography and bit-precise programming problems it has been designed for [10].

1.15 Defining functions

So far, we used Cryptol as a calculator: we typed in expressions and it gave us answers. This is great for experimenting, and exploring Cryptol itself. The next fundamental Cryptol idiom is the notion of a function. You have already used built-in functions `+`, `take`, etc. Of course, users can define their own functions as well. The recommended method is to define them in a file and load it, as in the next exercises.

Note: Reviewing the contents of Appendix E might help at this point. Especially the commands that will let you load files (`:l` and `:r`) in Cryptol.

Exercise 1.48. Type the following definition into a file and save it. Then load it in Cryptol and experiment.

```
increment : [8] -> [8]
increment x = x+1
```

In particular, try the following invocations:

[5] Another complaint is that "strong types are for weak minds." We do not disagree here: Cryptol programmers want to use the type system so we do not have to think as hard about writing correct code as we would without strong types.


```
increment 3
increment 255
increment 912
```

Do you expect the last call to type-check?

Note: We do not have to parenthesize the argument to `increment`, as in `increment(3)`. Function application is simply juxtaposition in Cryptol. However, you can write the parentheses if you want to, and you must use parentheses if you want to pass a negative argument, e.g. `increment(-2)` (recall exercise 1.38).

1.15.1 Definitions in the interpreter with `let`

As an alternative to creating and loading files, the Cryptol interpreter also supports simple definitions with `let`. Unlike file-based definitions, `let`-definitions are not persistent: They exist only until the same name is redefined or the interpreter exits with `:q`.

```
Cryptol> let r = {a = True, b = False}
Cryptol> r.a
True
```

Simple functions can be defined in the interpreter with `let`, with some restrictions: They must be entered on a single line, and they cannot have a separate type signature. If we want a function to have a more specific type than the inferred one, we must add type annotations to variables or other parts of the body of the definition.

Exercise 1.49. Enter the following definition into the Cryptol interpreter and check its type with `:t`.

```
let increment x = x+1
```

Modify the `let`-definition of `increment` so that it will have type `[8] -> [8]`.

1.15.2 Local names: `where` clauses

You can create local bindings in a `where` clause, to increase readability and give names to common subexpressions.

Exercise 1.50. Define and experiment with the following function:

```
twoPlusXY : ([8], [8]) -> [8]
twoPlusXY (x, y) = 2 + xy
   where xy = x * y
```

What is the signature of the function telling us?

Note: When calling `twoPlusXY`, you do need to parenthesize the arguments. But this is because you are passing it a tuple! The parentheses there are not for the application but rather in construction of the argument tuple. Cryptol does not automatically convert between tuples and curried application like in some other programming languages (e.g., one cannot pass a pair of type `(a, b)` to a function with type `a -> b -> c`).

Exercise 1.51. Comment out the type signature of `twoPlusXY` defined above, and let Cryptol infer its type. What is the inferred type? Why?

Exercise 1.52. Define a function with the following signature:

```
minMax4 : {a} (Cmp a) => [4]a -> (a, a)
```

such that the first element of the result is the minimum and the second element is the maximum of the given four elements. What happens when you try:

```
minMax4 [1 .. 4]
minMax4 [1 .. 5]
```

Why do we need the `(Cmp a)` constraint?

Exercise 1.53. Define a function `butLast` that returns all the elements of a given non-empty sequence, except for the last. How can you make sure that `butLast` can never be called with an empty sequence? (**Hint** You might find the Cryptol primitive functions `reverse` and `tail` useful.)

1.15.3 λ-expressions

One particular use case of a `where` clause is to introduce a helper function. If the function is simple enough, though, it may not be worth giving it a name. A λ-expression fits the bill in these cases, where you can introduce an unnamed function as an expression. The syntax differs from ordinary definitions in two minor details: instead of the name we use the backslash or "whack" character, '\', and the equals sign is replaced by an arrow '`->`'. (Since these functions do not have explicit names, they are sometimes referred to as "anonymous functions" as well. We prefer the term λ-expression, following the usual functional programming terminology [13].)

Below is an example of a λ-expression, allowing us to write functions inline:

```
Cryptol> f 8 where f x = x+1
9
Cryptol> (\x -> x+1) 8
9
```

λ-expressions are especially handy when you need to write small functions at the command line while interacting with the interpreter.

1.15.4 Using `zero` in functions

The constant `zero` comes in very handy in Cryptol whenever we need a polymorphic shape that consists of all `False` bits. The following two exercises utilize `zero` to define the functions `all` and `any` which, later in this book, you will find are very helpful functions for producing boolean values from a sequence.

Exercise 1.54. Write a function `all` with the following signature:

```
all : {n, a} (fin n) => (a -> Bit) -> [n]a -> Bit
```

such that `all f xs` returns `True` if all the elements in the sequence `xs` yield `True` for the function `f`. (**Hint** Use a complemented `zero`.) You should see:

```
Cryptol> all eqTen [10, 10, 10, 10] where eqTen x = x == 10
True
Cryptol> all eqTen [10, 10, 10, 5] where eqTen x = x == 10
False
```

(The `where` clause introduces a local definition that is in scope in the current expression. We have seen the details in section 1.15.) What is the value of `all f []` for an arbitrary function `f`? Is this reasonable?

Exercise 1.55. Write a function `any` with the following signature:

```
any : {n, a} (fin n) => (a -> Bit) -> [n]a -> Bit
```

such that `any f xs` returns `True` if any the elements in the sequence `xs` yield `True` for the function `f`. What is the value of `any f []`? Is this reasonable?

1.16 Recursion and recurrences

Cryptol allows both recursive function and value definitions. A recursive function is one that calls itself in its definition. Cryptol also allows the more general form of mutual recursion, where multiple functions can call each other in a cyclic fashion.

Exercise 1.56. Define two functions `isOdd` and `isEven` that each take a finite arbitrary sized word and returns a `Bit`. The functions should be mutually recursive. What extra predicates do you have to put on the size?

Exercise 1.57. While defining `isOdd` and `isEven` mutually recursively demonstrates the concept of recursion, it is not the best way of coding these two functions in Cryptol. Can you implement them using a constant time operation? (**Hint** What is the least significant bit of an even number? How about an odd one?)

Recurrences While Cryptol does support recursion, the explicit recursive function style is typically discouraged: Arbitrary recursion is hard to compile down to hardware. A much better notion is that of *recurrences*. A recurrence is a way of cyclically defining a value, typically a stream. It turns out that most recursive functions can be written in a recurrence style as well, something that might first come as a surprise. In particular, most recursive definitions arise from recurrence equations in cryptographic and data flow style programs, and Cryptol's comprehensions can succinctly represent these computations.

Exercise 1.58. In this exercise, we will define a function to compute the maximum element in a given sequence of numbers. Define the following function and load it into Cryptol:

```
maxSeq xs = ys ! 0
    where ys = [0] # [ max x y | x <- xs
                               | y <- ys
                     ]
```

What is the type of `maxSeq`? Try out the following calls:

```
maxSeq []
maxSeq [1 .. 10]
maxSeq ([10, 9 .. 1] # [1 .. 10])
```

Patterns of recurrence The definition pattern for `ys` in the definition of `maxSeq` above is very common in Cryptol, and it is well worth understanding it clearly. The basic idea is to create a sequence of running results, for each prefix of the input.

Exercise 1.59. Define a variant of `maxSeq` (let us call it `maxSeq'`) which returns the sequence `ys`, instead of its last element.

Running results It is very instructive to look at the results returned by `maxSeq'` that you have just defined in exercise 1.59:

```
Cryptol> maxSeq' []
[0]
Cryptol> maxSeq' [1 .. 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Cryptol> maxSeq' [10, 9 .. 1]
[0, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
Cryptol> maxSeq' [1, 3, 2, 4, 3, 5, 4, 7, 6, 8, 5]
[0, 1, 3, 3, 4, 4, 5, 5, 7, 7, 8, 8]
```

We clearly see the running results as they accumulate in `ys`. For the empty sequence, it only has `[0]` in it. For the monotonically increasing sequence `[1 .. 10]`, the maximum value keeps changing at each point, as each new element of `xs` is larger than the previous running result. When we try the sequence that always goes down (`[10, 9 .. 1]`), we find that the running maximum never changes after the first. The mixed input in the last call clearly demonstrates how the execution proceeds, the running maximum changing depending on the next element of `xs` and the running maximum so far. In the `maxSeq` function of exercise 1.58, we simply project out the last element of this sequence, obtaining the maximum of all elements in the given sequence.

Folds The pattern of recurrence employed in `maxSeq` is an instance of what is known as a *fold* [1]. Expressed in Cryptol terms, it looks as follows:

```
ys = [i] # [ f (x, y) | x <- xs
                      | y <- ys
           ]
```

where `xs` is some input sequence, `i` is the result corresponding to the empty sequence, and `f` is a transformer to compute the next element, using the previous result and the next input. This pattern can be viewed as generating a sequence of running values, accumulating them in `ys`. To illustrate, if `xs` is a sequence containing the elements [x_1, x_2, x_3 .. x_n], then successive elements of `ys` will be:

y_0 = i
y_1 = f(x_1, i)
y_2 = f(x_2, y_1)
y_3 = f(x_3, y_2)
...
y_n = f(x_n, y_{n-1})

Note how each new element of `ys` is computed by using the previous element and the next element of the input. The value `i` provides the seed. Consequently, `ys` will have one more element than `xs` does.

While loops An important use case of the above pattern is when we are interested in the final value of the accumulating values, as in the definition of `maxSeq`. When used in this fashion, the execution is reminiscent of a simple while loop that you might be familiar with from other languages, such as C:

```
-- C-like Pseudo-code!
y = i;                // running value, start with i
idx = 0;              // walk through the xs "array" using idx
while(idx < length (xs)) {
  y = f (xs[idx], y);  // compute next elt using the previous
  ++idx;
}
return y;
```

Note: If the while-loop analogy does not help you, feel free to ignore it. It is not essential. The moral of the story is this: if you feel like you need to write a while-loop in Cryptol to compute a value dependent upon the values in a datatype, you probably want to use a fold-like recurrence instead.

Exercise 1.60. Define a function that sums up a given sequence of elements. The type of the function should be:

```
sumAll : {n, a} (fin n, fin a) => [n][a] -> [a]
```

(**Hint** Use the folding pattern to create a sequence containing the partial running sums. What is the last element of this sequence?) Try it out on the following examples:

```
sumAll []
sumAll [1]
sumAll [1, 2]
sumAll [1, 2, 3]
sumAll [1, 2, 3, 4]
sumAll [1, 2, 3, 4, 5]
sumAll [1 .. 100]
```

Be mindful that if you do not specify the width of the result that you may get unexpected answers.

Exercise 1.61. Define a function `elem` with the following signature:


```
elem : {n, t} (fin n, Eq t) => (t, [n]t) -> Bit
```

such that `elem (x, xs)` returns `True` if `x` appears in `xs`, and `False` otherwise.[6]

Generalized folds The use of fold we have seen above is the simplest use case for recurrences in Cryptol. It is very common to see Cryptol programs employing some variant of this idea for most of their computations.

Exercise 1.62. Define the sequence of Fibonacci numbers `fibs`, so that `fibs @ n` is the n^{th} Fibonacci number [18]. You can assume 32 bits is sufficient for representing these fast growing numbers. (**Hint** Use a recurrence where the seed consists of two elements.)

1.17 Stream equations

Most cryptographic algorithms are described in terms of operations on bit-streams. A common way of depicting operations on bit-streams is using a *stream equation*, as shown in Figure 1.1:

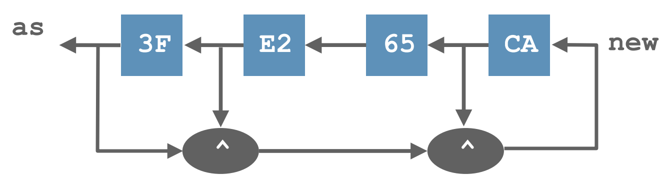

Figure 1.1: Equation for producing a stream of `as`

In this diagram the stream is seeded with four initial values (`3F, E2, 65, CA`). The subsequent elements (`new`) are appended to the stream, and are computed by xor-ing the current stream element with two additional elements extracted from further into the stream. The output from the stream is a sequence of values, known as *a*s.

The Cryptol code corresponding to this stream equation is:

```
as  = [0x3F, 0xE2, 0x65, 0xCA] # new
  where
    new = [ a ^ b ^ c | a <- as
                      | b <- drop`{1} as
                      | c <- drop`{3} as ]
```

Exercise 1.63. Write the Cryptol code corresponding to the stream equation in Figure 1.2:

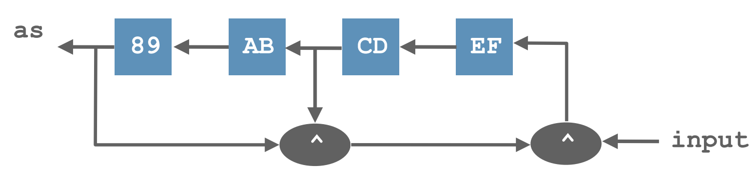

Figure 1.2: Equation for producing a stream of `as` from an initial seed and an input stream.

1.18 Type synonyms

Types in Cryptol can become fairly complicated, especially in the presence of records. Even for simple types, meaningful names should be used for readability and documentation. Type synonyms allow users to give names to arbitrary types.

[6]Note: this function is already present in the Cryptol prelude.

In this sense, they are akin to `typedef` declarations in C [8]. However, Cryptol's type synonyms are significantly more powerful than C's `typedef`s, since they can be parameterized by other types, much like in Haskell [13].

Here are some simple type synonym definitions:

```
type Word8       = [8]
type CheckedWord = (Word8, Bit)
type Point a     = {x : [a], y : [a]}
```

Type synonyms are either unparameterized (as in `Word8` and `CheckedWord`, or parameterized with other types (as in `Point`). Synonyms may depend upon other synonyms, as in the `CheckedWord` example. Once the synonym is given, it acts as an additional name for the underlying type, making it much easier to read and maintain.

For instance, we can write the function that returns the x-coordinate of a point as follows:

```
xCoord : {a} Point a -> [a]
xCoord p = p.x
```

Type synonyms look like distinct types when they are printed in the output of the `:t` and `:browse` commands. However, declaring a type synonym does not actually introduce a new *type*; it merely introduces a new *name* for an existing type. When Cryptol's type checker compares types, it expands all type synonyms first. So as far as Cryptol is concerned, `Word8` and `[8]` are the *same* type. Cryptol preserves type synonyms in displayed types as a convenience to the user.

For example, consider the following declarations:

```
type Word8     = [8]
type Word8'    = [8]
type B         = Word8
type A         = B
type WordPair  = (Word8, Word8')
type WordPair' = (Word8', Word8)

foo : Word8 -> Bit
foo x = True

bar : Word8' -> Bit
bar x = foo x
```

Within this type context, six different type *names* are declared, but there are not six distinct *types*. The first four (`Word8`, `Word8'`, `B`, and `A`) are all interchangeable abbreviations for `[8]`. The last two are synonymous and interchangeable with the pair type `([8], [8])`. Likewise, the function types of `foo` and `bar` are identical; thus `bar` can call `foo`.

Exercise 1.64. Define a type synonym for 3-dimensional points and write a function to determine if the point lies on any of the 3 axes.

Predefined type synonyms The following type synonyms are predefined in Cryptol:

```
type Bool = Bit
type Char = [8]
type String n = [n]Char
type Word n = [n]
```

For instance, a `String n` is simply a sequence of precisely *n* 8-bit words.

1.19 Type classes

Type classes are a way of describing behaviors shared by multiple types. As an example, consider the type of the function `==`:

```
Cryptol> :t (==)
(==) : {a} (Eq a) => a -> a -> Bit
```

This type signature may be read as, "equality is an operator that takes two objects of any single type that can be compared and returns a Bit."

Cryptol defines a collection of basic type classes: `Logic`, `Zero`, `Eq`, `Cmp`, `SignedCmp`, `Ring`, `Integral`, `Field`, `Round`, and `Literal`. These appear in the type signature of operators and functions that require them. If a function you define calls, for example, `+`, on two arguments both of type `a`, the type constraints for `a` will include `(Ring a)`.

- The `Logic` typeclass includes the binary logical operators `&&`, `||`, and `^`, as well as the unary operator `~`. Cryptol types made of bits (but not those containing unbounded integers) are instances of class `Logic`.
- The `Zero` typeclass includes the special constant `zero`. The shifting operators `<<` and `>>` are also in class `Zero`, because they can shift in zero values. All of the built-in types of Cryptol are instances of class `Zero`.
- The `Eq` typeclass includes the equality testing operations `==` and `!=`. Function types are not in class `Eq` and cannot be compared with `==`, but Cryptol does provide a special pointwise comparison operator for functions, `(===) : {a b} (Cmp b) => (a -> b) -> (a -> b) -> a -> Bit`.
- The `Cmp` typeclass includes the binary relation operators `<`, `>`, `<=`, `>=`, as well as the binary functions `min` and `max`.
- The `SignedCmp` typeclass includes the binary relation operators `<$`, `>$`, `<=$`, and `>=$`. These are like `<`, `>`, `<=`, and `>=`, except that they interpret bitvectors as *signed* 2's complement numbers, whereas the `Cmp` operations use the *unsigned* ordering. `SignedCmp` does not contain the other atomic numeric types in Cryptol, just bitvectors.
- The `Ring` typeclass includes the binary operators `+`, `-`, `*`, and the unary operators `negate` and `fromInteger`. All the numeric types in Cryptol are members of `Ring`.
- The `Integral` typeclass represents values that are "like integers". It inclues the integral division and modulus operators `/` and `%`, and the `toInteger` casting function. The sequence indexing, update and shifting operations take index arguments that can be of any `Integral` type. Infinite sequence enumerations `[x ...]` and `[x, y ...]` are also defined for class `Integral`. Bitvectors and integers members of `Integral`.
- The `Field` typeclass represents values that, in addition to being a `Ring`, have multiplictive inverses. It includes the field division operation `/.` and the `recip` operation for computing the reciprocol of a value. Currently, only type `Rational` is a member of this class.
- The `Round` typeclass contains types that can be rounded to integers. It includes `floor`, `ceiling`, `trunc`, `roundAway` and `roundToEven` operations, which are all different ways of rounding values to integers. Currently, only type `Rational` is a member of this class.
- The `Literal` typeclass includes numeric literals.

Exercise 1.65. Without including an explicit type declaration, define a function that Cryptol infers has the following type:

```
cmpRing : {a,b} (Eq a, Ring b) => a -> a -> b -> b
```

1.20 Type vs. value variables

Its powerful type system is one of the key features of Cryptol. We have encountered many aspects of types already. You may have noticed, in functions such as `split`, that when you call a function in Cryptol, there are two kinds of parameters you can pass: *value variables* and *type variables*.

Consider the `split` function that we previously examined in Exercise 1.46. Recall that `split`'s type is:

```
split : {parts, each, a} (fin each) =>
          [parts * each]a -> [parts][each]a
```

When applying `split`, one typically specifies a concrete value for the formal parameter `parts`:

```
Cryptol> split`{parts=3} [1..12]
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

In this example, the term `` `{parts=3} `` passes `3` to the `parts` type variable argument, and the `[1..12]` is passing a sequence as the first (and only) *value argument*.

A *value variable* is the kind of variable you are used to from normal programming languages. This kind of variable represents a normal run-time value.

A *type variable*, on the other hand, allows you to express interesting (arithmetic) constraints on *types*. These variables express things like lengths of sequences or relationships between lengths of sequences. Type variable values are computed statically—they never change at runtime.[7]

1.20.1 Positional vs. named type arguments

Cryptol permits type variables to be passed either by name (as in `` `{parts=3} `` above), or by position (leaving out the name). For functions you define, the position is the order in which the type variables are declared in your function's type signature. If you are not sure what that is, you can always use the `:t` command to find out the position of type variables.

For example:

```
Cryptol> :t groupBy
groupBy : {each, parts, elem}
          (fin each) => [parts * each]elem
                     -> [parts][each]elem
```

tells us that that `parts` is in the second position of `groupBy`'s type signature, so the positional-style call equivalent to our example is:

```
Cryptol> groupBy`{_,3}[1..12]
```

Note the use of an underscore in order to pass `3` in the second position. Positional arguments are most often used when the type argument is the first argument and when the name of the argument does not add clarity. The `` groupBy`{_,3} `` is not as self-explanatory as `` groupBy`{parts=3} ``. On the other hand, our use of positional arguments to `take` in previous chapters is very clear, as in:

```
Cryptol> take`{3}[1..12]
[1, 2, 3]
```

Tip: Cryptol programs that use named arguments are more maintainable and robust during program evolution. E.g., you can reorder parameters or refactor function definitions much more easily if invocations of those functions use named, rather than positional, arguments.

1.20.2 Type context vs. variable context

You have seen, in the discussion of type variables above, that Cryptol has two kinds of variables—type variables and value variables. Type variables normally show up in type signatures, and value variables normally show up in function definitions. Sometimes you may want to use a type variable in a context where value variables would normally be used. To do this, use the backtick character `` ` ``.

The definition of the built-in `length` function is a good example of the use of backtick:

```
length : {n, a, b} (fin n, Literal n b) => [n]a -> b
length _ = `n
```

[7]In this way, they are similar (but more expressive than) templates in languages like C++ or Java. If you want to learn more about this area, look up the term "type-level naturals".

Tip: Note there are some subtle things going on in the above definition of `length`. First, arithmetic constraints on types are position-independent; properties of formal parameters early in a signature can depend upon those late in a signature. Second, type constraints can refer to not only other functions, but recursively to the function that is being defined (either directly, or transitively).

Type constraints can get pretty crazy in practice, especially deep in the signature of crypto code subsystems. Our suggestion is that you should not chase the dragon's tail of feedback from the typechecker in attempting to massage your specification's types for verification. Instead, think carefully about the meaning and purpose of the concepts in your specification, introduce appropriate type synonyms, and ensure that the specification is clear and precise. Trust that the interpreter and the verifier will do the right thing.

The bounds in a finite sequence literal (such as `[1 .. 10]`) in Cryptol are type-level values because the length of a sequence is part of its type. Only type-level values can appear in a finite sequence definition. You cannot write `[a .. b]` where either `a` or `b` are arguments to a function. On the other hand, an infinite sequence's type is fixed (`[inf]a`), so the initial value in an infinite sequence can be a runtime variable or a type variable, but type variables are escaped here using a `` ` ``.

This is probably obvious, but there is no way to get a value variable to appear in a type context. Types must be known at "compile time," and (non-literal) values are not, so there is no way to use them in that way.

1.20.3 Inline argument type declarations

So far when we have defined a function, we have declared the type of its arguments and its return value in a separate type declaration. When you are initially writing code, you might not know exactly what a function's full type is (including the constraints), but you may know (and need to express) the types of the function's arguments. Cryptol's syntax for this should look familiar:

```
addBytes (x:[8]) (y:[8]) = x + y
```

This defines a function that takes two bytes as input, and returns their sum. Note that the use of parentheses `( )` is mandatory.

Here is a more interesting example:

```
myWidth (x:[w]a) = `w
```

1.21 Type constraint synonyms

Sometimes programs have certain combinations of type constraints that appear over and over in many places. For convenience, Cryptol allows the programmer to declare named sets of type constraints that can be used in other function type signatures. Typically a named type constraint will have one or more type parameters. The syntax is very similar to a type synonym declaration:

```
type constraint myConstraint a b = (fin a, fin b, b >= width a)
```

Wherever a type constraint synonym is used, it is as if its definition is expanded in place. So the following two signatures would be equivalent:

```
myWidth : {bits,len,elem} (fin len, fin bits, bits >= width len) => [len] elem -> [bits]
myWidth : {bits,len,elem} (myConstraint len bits) => [len] elem -> [bits]
```

1.22 Program structure with modules

When a cryptographic specification gets very large it can make sense to decompose its functions into modules. Doing this well encourages code reuse, so it's a generally good thing to do. Cryptol's module system is simple and easy to use. Here's a quick overview:

A module's name should be the same as the filename the module is defined in, and each file may contain only a single module. For example, the `utilities` module should be defined in a file called `utilities.cry`. To specify that a file defines a module, its first non-commented line should be:


```
module utilities where
```

After that the variables and functions you define will be contained (in this example) in the *utilities* module.

In the code where you want to use a module, you `import` it like this:

```
import utilities
```

Cryptol will look for the file `utilities.cry` in the current directory. Once you've imported a module, all of its variables and functions are available to use in your code.

If you're writing a module that has both *private* and *public* definitions, you can hide the ones that shouldn't be exported to modules that include it by using the `private` keyword, like this:

```
private internalDouble x = x + x
exportedDouble = x * 2
```

As you can tell, by default definitions are exported to including modules.

For a large project it can be convenient to place modules in a directory structure. In this case, the directory structure becomes part of the modules' names. For example, when placing `SHA3.cry` in the `Hash` directory and accessing it from `HMAC.cry` you would need to name the modules accordingly:

```
sha3 : {n} (fin n) => [n] -> [512]
sha3 = error "Stubbed, for demonstration only: sha3-512"

blocksize : {n} (fin n, n >= 10) => [n]
blocksize = 576

module Hash::SHA3 where
import Hash::SHA3

hmac : {keySize, msgSize} (fin keySize, fin msgSize) => [keySize] -> [msgSize] -> [512]
hmac k m = sha3 (ko # sha3 (ki # m))
 where ko    = zipWith (^) kFull (join (repeat 0x5c))
       ki    = zipWith (^) kFull (join (repeat 0x36))
       kFull = if `keySize == blocksize
                 then take (k#zero)
                 else sha3 k
```

Finally, if you're importing a module that defines something with a name that you would like to define in your code, you can do a *qualified* import of that module like this:

```
import utilities as util
```

Now, instead of all the definitions being available in your module, they are qualified with the name you provided, in this case `util`. This means you will prefix those names with `util::` when you call them, and the unqualified names are able to be defined in your own code.

```
import utilities as util
// let's say utililities.cry defines "all", and we want to use
// it in our refined definition of all:
all xs = util::all xs && (length xs) > 0
```

1.23 The road ahead

In this introductory chapter, we have seen essentially all of the language elements in Cryptol. The concepts go deeper, of course, but you now have enough knowledge to tackle large Cryptol programming tasks. As with any new language, the more exercises you do, the more you will feel comfortable with the concepts. In fact, we will take that point of view in the remainder of this document to walk you through a number of different examples (both small and large), employing the concepts we have seen thus far.

Chapter 2

Classic ciphers

Modern cryptography has come a long way. In his excellent book on cryptography, Singh traces it back to at least 5th century B.C., to the times of Herodotus and the ancient Greeks [14]. That's some 2500 years ago, and surely we do not use those methods anymore in modern day cryptography. However, the basic techniques are still relevant for appreciating the art of secret writing.

Shift ciphers construct the ciphertext from the plaintext by means of a predefined *shifting* operation, where the cipherkey of a particular shift algorithm defines the shift amount of the cipher. Transposition ciphers work by keeping the plaintext the same, but *rearrange* the order of the characters according to a certain rule. The cipherkey is essentially the description of how this transposition is done. Substitution ciphers generalize shifts and transpositions, allowing one to substitute arbitrary codes for plaintext elements. In this chapter, we will study several examples of these techniques and see how we can code them in Cryptol.

In general, ciphers boil down to pairs of functions *encrypt* and *decrypt* which "fit together" in the appropriate way. Arguing that a cryptographic function is *correct* is subtle.

Correctness of cryptography is determined by cryptanalyses by expert cryptographers. Each kind of cryptographic primitive (i.e., a hash, a symmetric cipher, an asymmetric cipher, etc.) has a set of expected properties, many of which can only be discovered and proven by hand through a lot of hard work. Thus, to check the correctness of a cryptographic function, a best practice for Cryptol use is to encode as many of these properties as one can in Cryptol itself and use Cryptol's validation and verification capabilities, discussed later in chapter 4. For example, the fundamental property of most ciphers is that encryption and decryption are inverses of each other.

To check the correctness of an *implementation* I of a cryptographic function C means that one must show that the implementation I behaves as the specification (C) stipulates. In the context of cryptography, the minimal conformance necessary is that I's output *exactly* conforms to the output characterized by C. But just because a cryptographic implementation is *functionally correct* does not mean it is *secure*. The subtleties of an implementation can leak all kinds of information that harm the security of the cryptography, including abstraction leaking of sensitive values, timing attacks, side-channel attacks, etc. These kinds of properties cannot currently be expressed or reasoned about in Cryptol.

Also, Cryptol does *not* give the user any feedback on the *strength* of a given (cryptographic) algorithm. While this is an interesting and useful feature, it is not part of Cryptol's current capabilities.

2.1 Caesar's cipher

Caesar's cipher (a.k.a. Caesar's shift) is one of the simplest ciphers. The letters in the plaintext are shifted by a fixed number of elements down the alphabet. For instance, if the shift is 2, `A` becomes `C`, `B` becomes `D`, and so on. Once we run out of letters, we circle back to `A`; so `Y` becomes `A`, and `Z` becomes `B`. Coding Caesar's cipher in Cryptol is quite straightforward (recall from section 1.18 that a `String n` is simply a sequence of n 8-bit words.):

```
caesar : {n} ([8], String n) -> String n
caesar (s, msg) = [ shift x | x <- msg ]
      where map     = ['A' .. 'Z'] <<< s
            shift c = map @ (c - 'A')
```

In this definition, we simply get a message `msg` of type `String n`, and perform a `shift` operation on each one of the elements. The `shift` function is defined locally in the `where` clause. To compute the shift, we first find the distance of the letter from the character `'A'` (via `c - 'A'`), and look it up in the mapping imposed by the shift. The `map` is simply the alphabet rotated to the left by the shift amount, `s`. Note how we use the enumeration `['A' .. 'Z']` to get all the letters in the alphabet.

Exercise 2.1. What is the map corresponding to a shift of 2? Use Cryptol's `<<<` to compute it. You can use the command `:set ascii=on` to print strings in ASCII, like this:

```
Cryptol> :set ascii=on
Cryptol> "Hello World"
"Hello World"
```

Why do we use a left-rotate, instead of a right-rotate?

Exercise 2.2. Use the above definition to encrypt the message `"ATTACKATDAWN"` by shifts 0, 3, 12, and 52. What happens when the shift is a multiple of 26? Why?

Exercise 2.3. Write a function `dCaesar` which will decrypt a ciphertext constructed by a Caesar's cipher. It should have the same signature as `caesar`. Try it on the examples from the previous exercise.

Exercise 2.4. Observe that the shift amount in a Caesar cipher is very limited: Any shift of `d` is equivalent to a shift by `d % 26`. (For instance shifting by 12 and 38 is the same thing, due to wrap around at 26.) Based on this observation, how strong do you think the Caesar's cipher is? Describe a simple attack that will recover the plaintext and automate it using Cryptol. Use your function to crack the ciphertext `JHLZHYJPWOLYPZDLHR`.

Exercise 2.5. One classic trick to strengthen ciphers is to use multiple keys. By repeatedly encrypting the plaintext multiple times we can hope that it will be more resistant to attacks. Do you think this scheme might make the Caesar cipher stronger?

Exercise 2.6. What happens if you pass `caesar` a plaintext that has non-uppercase letters in it? (Let's say a digit.) How can you fix this deficiency?

2.2 Vigenère cipher

The Vigenère cipher is a variation on the Caesar's cipher, where one uses multiple shift amounts according to a keyword [23]. Despite its simplicity, it earned the notorious description *le chiffre indèchiffrable* ("the indecipherable cipher" in French), as it was unbroken for a long period of time. It was very popular in the 16th century and onwards, only becoming routinely breakable by mid-19th century or so.

To illustrate the operation of the Vigenère cipher, let us consider the plaintext `ATTACKATDAWN`. The cryptographer picks a key, let's say `CRYPTOL`. We line up the plaintext and the key, repeating the key as much as as necessary, as in the top two lines of the following:

Plaintext :	`ATTACKATDAWN`
Cipherkey :	`CRYPTOLCRYPT`
Ciphertext:	`CKRPVYLVUYLG`

We then proceed pair by pair, shifting the plaintext character by the distance implied by the corresponding key character. The first pair is `A-C`. Since `C` is two positions away from `A` in the alphabet, we shift `A` by two positions, again obtaining `C`. The second pair `T-R` proceeds similarly: Since `R` is 17 positions away from `A`, we shift `T` down 17 positions, wrapping around `Z`, obtaining `K`. Proceeding in this fashion, we get the ciphertext `CKRPVYLVUYLG`. Note how each step of the process is a simple application of the Caesar's cipher.

Exercise 2.7. One component of the Vigenère cipher is the construction of the repeated key. Write a function `cycle` with the following signature:

```
cycle : {n, a} (fin n, n >= 1) => [n]a -> [inf]a
```

such that it returns the input sequence appended to itself repeatedly, turning it into an infinite sequence. Why do we need the predicate `n >= 1`?

Exercise 2.8. Program the Vigenère cipher in Cryptol. It should have the signature:

```
vigenere : {n, m} (fin n, n >= 1) => (String n, String m) -> String m
```

where the first argument is the key and the second is the plaintext. Note how the signature ensures that the input string and the output string will have precisely the same number of characters, `m`. (**Hint** Use Caesar's cipher repeatedly.)

Exercise 2.9. Write the decryption routine for Vigenère. Then decode `"XZETGSCGTYCMGEQGAGRDEQC"` with the key `"CRYPTOL"`.

Exercise 2.10. A known-plaintext attack is one where an attacker obtains a plaintext-ciphertext pair, without the key. If the attacker can figure out the key based on this pair then he can break all subsequent communication until the key is replaced. Describe how one can break the Vigenère cipher if a plaintext-ciphertext pair is known.

2.3 The atbash

The atbash cipher is a form of a shift cipher, where each letter is replaced by the letter that occupies its mirror image position in the alphabet. That is, `A` is replaced by `Z`, `B` by `Y`, etc. Needless to say the atbash is hardly worthy of cryptographic attention, as it is trivial to break.

Exercise 2.11. Program the atbash in Cryptol. What is the code for `ATTACKATDAWN`?

Exercise 2.12. Program the atbash decryption in Cryptol. Do you have to write any code at all? Break the code `ZGYZHSRHHVOUWVXIBKGRMT`.

2.4 Substitution ciphers

Substitution ciphers generalize all the ciphers we have seen so far, by allowing arbitrary substitutions to be made for individual "components" of the plaintext [22]. Note that these components need not be individual characters, but rather can be pairs or even triples of characters that appear consecutively in the text. (The multi-character approach is termed *polygraphic*.) Furthermore, there are variants utilizing multiple *polyalphabetic* mappings, as opposed to a single *monoalphabetic* mapping. We will focus on monoalphabetic simple substitutions, although the other variants are not fundamentally more difficult to implement.

Tip: For the exercises in this section we will use a running key repeatedly. To simplify your interaction with Cryptol, put the following definition in your program file:

```
substKey : String 26
substKey = "FJHWOTYRXMKBPIAZEVNULSGDCQ"
```

The intention is that `substKey` maps `A` to `F`, `B` to `J`, `C` to `H`, and so on.

Exercise 2.13. Implement substitution ciphers in Cryptol. Your function should have the signature:

```
subst : {n} (String 26, String n) -> String n
```

where the first element is the key (like `substKey`). What is the code for `"SUBSTITUTIONSSAVETHEDAY"` for the key `substKey`?

Decryption Programming decryption is more subtle. We can no longer use the simple selection operation (`@`) on the key. Instead, we have to search for the character that maps to the given ciphertext character.

Exercise 2.14. Write a function `invSubst` with the following signature:

```
invSubst : (String 26, Char) -> Char
```

such that it returns the mapped plaintext character. For instance, with `substKey`, `F` should get you `A`, since the key maps `A` to `F`:

```
Cryptol> invSubst (substKey, 'F')
A
```

And similarly for other examples:

```
Cryptol> invSubst (substKey, 'J')
B
Cryptol> invSubst (substKey, 'C')
Y
Cryptol> invSubst (substKey, 'Q')
Z
```

One question is what happens if you search for a non-existing character. In this case you can just return `0`, a non-valid ASCII character, which can be interpreted as *not found*.

Hint. *Use a fold (see page 20).*

Exercise 2.15. Using `invSubst`, write the decryption function `dSubst`. It should have the exact same signature as `subst`. Decrypt `FUUFHKFUWFGI`, using our running key.

Exercise 2.16. Try the substitution cipher with the key `AAAABBBBCCCCDDDDEEEEFFFFGG`. Does it still work? What is special about `substKey`?

2.5 The scytale

The scytale is one of the oldest cryptographic devices ever, dating back to at least the first century A.D. [21]. Ancient Greeks used a leather strip on which they would write their plaintext message. The strip would be wrapped around a rod of a certain diameter. Once the strip is completely wound, they would read the text row-by-row, essentially transposing the letters and constructing the ciphertext. Since the ciphertext is formed by a rearrangement of the plaintext, the scytale is an example of a transposition cipher. To decrypt, the ciphertext needs to be wrapped around a rod of the same diameter, reversing the process. The cipherkey is essentially the diameter of the rod used. Needless to say, the scytale does not provide a very strong encryption mechanism.

Abstracting away from the actual rod and the leather strip, encryption is essentially writing the message column-by-column in a matrix and reading it row-by-row. Let us illustrate with the message `ATTACKATDAWN`, where we can fit 4 characters per column:

```
ACD
TKA
TAW
ATN
```

To encrypt, we read the message row-by-row, obtaining `ACDTKATAWATN`. If the message does not fit properly (i.e., if it has empty spaces in the last column), it can be padded by `Z`'s or some other agreed upon character. To decrypt, we essentially reverse the process, by writing the ciphertext row-by-row and reading it column-by-column.

Notice how the scytale's operation is essentially matrix transposition. Therefore, implementing the scytale in Cryptol is merely an application of the `transpose` function. All we need to do is group the message by the correct number of elements using `split`. Below, we define the `diameter` to be the number of columns we have. The type synonym `Message` ensures we only deal with strings that properly fit the "rod," by using `r` number of rows:

```
scytale : {row, diameter} (fin row, fin diameter)
          => String (row * diameter) -> String (diameter * row)
scytale msg = join (transpose msg')
     where   msg' : [diameter][row][8]
             msg' = split msg
```

The signature on `msg'` is revealing: We are taking a string that has `diameter * row` characters in it, and chopping it up so that it has `row` elements, each of which is a string that has `diameter` characters in it. Here is Cryptol in action, encrypting the message `ATTACKATDAWN`, with `diameter` set to `3`:

```
Cryptol> :set ascii=on
Cryptol> scytale `{diameter=3} "ATTACKATDAWN"
"ACDTKATAWATN"
```

Decryption is essentially the same process, except we have to `split` so that we get `diameter` elements out:

```
dScytale : {row, diameter} (fin row, fin diameter)
           => String (row * diameter) -> String (diameter * row)
dScytale msg = join (transpose msg')
   where   msg' : [row][diameter][8]
           msg' = split msg
```

Again, the type on `msg'` tells Cryptol that we now want `diameter` strings, each of which is `row` long. It is important to notice that the definitions of `scytale` and `dScytale` are precisely the same, except for the signature on `msg'`! When viewed as a matrix, the types precisely tell which transposition we want at each step. We have:

```
Cryptol> dScytale `{diameter=3} "ACDTKATAWATN"
"ATTACKATDAWN"
```

Exercise 2.17. What happens if you comment out the signature for `msg'` in the definition of `scytale`? Why?

Exercise 2.18. How would you attack a scytale encryption, if you don't know what the diameter is?

Chapter 3

The Enigma machine

The Enigma machine is probably the most famous of all cryptographic devices in history, due to the prominent role it played in WWII [16]. The first Enigma machines were available around 1920s, with various models in the market for commercial use. When Germans used the Enigma during WWII, they were using a particular model referred to as the *Wehrmacht Enigma*, a fairly advanced model available at the time.

The most important role of Enigma is in its role in the use of automated machines to aid in secret communication, or what is known as *mechanizing secrecy*. One has to understand that computers as we understand them today were not available when Enigma was in operation. Thus, the Enigma employed a combination of mechanical (keyboard, rotors, etc.) and electrical parts (lamps, wirings, etc.) to implement its functionality. However, our focus in this chapter will not be on the mechanical aspects of Enigma at all. For a highly readable account of that, we refer the reader to Singh's excellent book on cryptography [14]. Instead, we will model Enigma in Cryptol in an algorithmic sense, implementing Enigma's operations without any reference to the underlying mechanics. More precisely, we will model an Enigma machine that has a plugboard, three interchangeable scramblers, and a fixed reflector.

3.1 The plugboard

Enigma essentially implements a polyalphabetic substitution cipher (section 2.4), consisting of a number of rotating units that jumble up the alphabet. The first component is the so called plugboard (*Steckerbrett* in German). In the original Enigma, the plugboard provided a means of interchanging 6 pairs of letters. For instance, the plugboard could be set-up so that pressing the `B` key would actually engage the `Q` key, etc. We will slightly generalize and allow any number of pairings, as we are not limited by the availability of cables or actual space to put them in a box! Viewed in this sense, the plugboard is merely a permutation of the alphabet. In Cryptol, we can represent the plugboard combination by a string of 26 characters, corresponding to the pairings for each letter in the alphabet from `A` to `Z`:

```
type Permutation = String 26
type Plugboard = Permutation
```

For instance, the plugboard matching the pairs `A-H`, `C-G`, `Q-X`, `T-V`, `U-Y`, `W-M`, and `O-L` can be created as follows:

```
plugboard : Plugboard
plugboard = "HBGDEFCAIJKOWNLPXRSVYTMQUZ"
```

Note that if a letter is not paired through the plugboard, then it goes untouched, i.e., it is paired with itself.

Exercise 3.1. Use Cryptol to verify that the above plugboard definition indeed implements the pairings we wanted.

Note: In Enigma, the plugboard pairings are symmetric; if `A` maps to `H`, then `H` must map to `A`.

3.2 Scrambler rotors

The next component of the Enigma are the rotors that scramble the letters. Rotors (*walzen* in German) are essentially permutations, with one little twist: as their name implies, they rotate. This rotation ensures that the next character the rotor will process will be encrypted using a different alphabet, thus giving Enigma its polyalphabetic nature.

The other trick employed by Enigma is how the rotations are done. In a typical setup, the rotors are arranged so that the first rotor rotates at every character, while the second rotates at every 26th, the third at every 676th ($= 26 * 26$), etc. In a sense, the rotors work like the odometer in your car, one full rotation of the first rotor triggers the second, whose one full rotation triggers the third, and so on. In fact, more advanced models of Enigma allowed for two notches per rotor, i.e., two distinct positions on the rotor that will allow the next rotor in sequence to rotate itself. We will allow ourselves to have any number of notches, by simply pairing each substituted letter with a bit value saying whether it has an associated notch:[1]

```
type Rotor = [26](Char, Bit)
```

The function `mkRotor` will create a rotor for us from a given permutation of the letters and the notch locations:[2]

```
mkRotor : {n} (fin n) => (Permutation, String n) -> Rotor
mkRotor (perm, notchLocations) = [ (p, elem (p, notchLocations))
                                 | p <- perm
                                 ]
```

Let us create a couple of rotors with notches:

```
rotor1, rotor2, rotor3 : Rotor
rotor1 = mkRotor ("RJICAWVQZODLUPYFEHXSMTKNGB", "IO")
rotor2 = mkRotor ("DWYOLETKNVQPHURZJMSFIGXCBA", "B")
rotor3 = mkRotor ("FGKMAJWUOVNRYIZETDPSHBLCQX", "CK")
```

For instance, `rotor1` maps `A` to `R`, `B` to `J`, `...`, and `Z` to `B` in its initial position. It will engage its notch if one of the permuted letters `I` or `O` appear in its first position.

Exercise 3.2. Write out the encrypted letters for the sequence of 5 `C`'s for `rotor1`, assuming it rotates in each step. At what points does it engage its own notch to signal the next rotor to rotate?

3.3 Connecting the rotors: notches in action

The original Enigma had three interchangeable rotors. The operator chose the order they were placed in the machine. In our model, we will allow for an arbitrary number of rotors. The tricky part of connecting the rotors is ensuring that the rotations of each are done properly.

Let us start with a simpler problem. If we are given a rotor and a particular letter to encrypt, how can we compute the output letter and the new rotor position? First of all, we will need to know if the rotor should rotate itself, that is if the notch between this rotor and the previous one was activated. Also, we need to find out if the act of rotation in this rotor is going to cause the next rotor to rotate. We will model this action using the Cryptol function `scramble`:

```
scramble : (Bit, Char, Rotor) -> (Bit, Char, Rotor)
```

The function `scramble` takes a triple `(rotate, c, rotor)`:

- `rotate`: if `True`, this rotor will rotate after encryption. Indicates that the notch between this rotor and the previous one was engaged,
- `c`: the character to encrypt, and
- `rotor`: the current state of the rotor.

Similarly, the output will also be a triple:

- `notch`: `True` if the notch on this rotor engages, i.e., if the next rotor should rotate itself,

[1] The type definition for `Char` was given in Example 2.4-2.14.
[2] The function `elem` was defined in exercise 1.61.

- `c'`: the result of encrypting (substituting) for `c` with the current state of the rotor.
- `rotor'`: the new state of the rotor. If no rotation was done this will be the same as `rotor`. Otherwise it will be the new substitution map obtained by rotating the old one to the left by one.

Coding `scramble` is straightforward:

```
scramble (rotate, c, rotor) = (notch, c', rotor')
  where
    (c', _)     = rotor @ (c - 'A')
    (_, notch) = rotor @ 0
    rotor'      = if rotate then rotor <<< 1 else rotor
```

To determine `c'`, we use the substitution map to find out what this rotor maps the given character to, with respect to its current state. Note how Cryptol's pattern matching notation helps with extraction of `c'`, as we only need the character, not whether there is a notch at that location. (The underscore character use, '`_`', means that we do not need the value at the position, and hence we do not give it an explicit name.) To determine if we have our notch engaged, all we need to do is to look at the first elements notch value, using Cryptol's selection operator (`@ 0`), and we ignore the permutation value there this time, again using pattern matching. Finally, to determine `rotor'` we merely rotate left by 1 if the `rotate` signal was received. Otherwise, we leave the `rotor` unchanged.

Exercise 3.3. Redo exercise 3.2, this time using Cryptol and the `scramble` function.

Note: The actual mechanics of the Enigma machine were slightly more complicated: due to the keyboard mechanism and the way notches were mechanically built, the first rotor was actually rotating before the encryption took place. Also, the middle rotor could double-step if it engages its notch right after the third rotor does [2].
We will take the simpler view here and assume that each key press causes an encryption to take place, *after* which the rotors do their rotation, getting ready for the next input. The mechanical details, while historically important, are not essential for our modeling purposes here. Also, the original Enigma had *rings*, a relatively insignificant part of the whole machine, that we ignore here.

Sequencing the rotors Now that we have the rotors modeled, the next task is to figure out how to connect them in a sequence. As we mentioned, Enigma had 3 rotors originally (later versions allowing 4). The three rotors each had a single notch (later versions allowing double notches). Our model allows for arbitrary number of rotors and arbitrary number of notches on each. The question we now tackle is the following: Given a sequence of rotors, how do we run them one after the other? We are looking for a function with the following signature:

```
joinRotors : {n} (fin n) => ([n]Rotor, Char) -> ([n]Rotor, Char)
```

That is, we receive `n` rotors and the character to be encrypted, and return the updated rotors (accounting for their rotations) and the final character. The implementation is an instance of the fold pattern (section 1.16), using the `scramble` function we have just defined:

```
joinRotors (rotors, inputChar) = (rotors', outputChar)
  where
    initRotor = mkRotor (['A' .. 'Z'], [])
    ncrs : [n+1](Bit, [8], Rotor)
    ncrs = [(True, inputChar, initRotor)]
              # [  scramble (notch, char, r)
                 | r <- rotors
                 | (notch, char, rotor') <- ncrs
               ]
    rotors' = tail [ r | (_, _, r) <- ncrs ]
    (_, outputChar, _) = ncrs ! 0
```

The workhorse in `joinRotors` is the definition of `ncrs`, a mnemonic for *notches-chars-rotors*. The idea is fairly simple. We simply iterate over all the given rotors (`r <- rotors`), and `scramble` the current character `char`, using the rotor `r` and the notch value `notch`. These values come from `ncrs` itself, using the fold pattern encoded by the comprehension. The only question is what is the seed value for this fold?

The seed used in `ncrs` is `(True, inputChar, initRotor)`. The first component is `True`, indicating that the very first rotor should always rotate itself at every step. The second element is `inputChar`, which is the input to the whole sequence of rotors. The only mysterious element is the last one, which we have specified as `initRotor`. This rotor is defined so that it simply maps the letters to themselves with no notches on it, by a call to the `mkRotor` function we have previously defined. This rotor is merely a place holder to kick off the computation of `ncrs`, it acts as the identity element in a sequence of rotors. To compute `rotors'`, we merely project the third component of `ncrs`, being careful about skipping the first element using `tail`. Finally, `outputChar` is merely the output coming out of the final rotor, extracted using `!0`. Note how we use Cryptol's pattern matching to get the second component out of the triple in the last line.

Exercise 3.4. Is the action of `initRotor` ever used in the definition of `joinRotors`?

Exercise 3.5. What is the final character returned by the expression:

```
joinRotors ([rotor1 rotor2 rotor3], 'F')
```

Use paper and pencil to figure out the answer by tracing the execution of `joinRotors` before running it in Cryptol!

3.4 The reflector

The final piece of the Enigma machine is the reflector (*umkehrwalze* in German). The reflector is another substitution map. Unlike the rotors, however, the reflector did not rotate. Its main function was ensuring that the process of encryption was reversible: The reflector did one final jumbling of the letters and then sent the signal back through the rotors in the *reverse* order, thus completing the loop and allowing the signal to reach back to the lamps that would light up. For our purposes, it suffices to model it just like any other permutation:

```
type Reflector = Permutation
```

Here is one example:

```
reflector : Reflector
reflector = "FEIPBATSCYVUWZQDOXHGLKMRJN"
```

Like the plugboard, the reflector is symmetric: If it maps `B` to `E`, it should map `E` to `B`, as in the above example. Furthermore, the Enigma reflectors were designed so that they never mapped any character to themselves, which is true for the above permutation as well. Interestingly, this idea of a non-identity reflector (i.e., never mapping any character to itself) turned out to be a weakness in the design, which the allies exploited in breaking the Enigma during WWII [14].

Exercise 3.6. Write a function `checkReflector` with the signature:

```
checkReflector : Reflector -> Bit
```

such that it returns `True` if a given reflector is good (i.e., symmetric and non-self mapping) and `False` otherwise. Check that our definition of `reflector` above is a good one. (**Hint** Use the `all` function you have defined in exercise 1.54.)

3.5 Putting the pieces together

We now have all the components of the Enigma: the plugboard, rotors, and the reflector. The final task is to implement the full loop. The Enigma ran all the rotors in sequence, then passed the signal through the reflector, and ran the rotors in reverse one more time before delivering the signal to the lamps.

Before proceeding, we will define the following two helper functions:

```
substFwd, substBwd : (Permutation, Char) -> Char
substFwd (perm, c) = perm @ (c - 'A')
substBwd (perm, c) = invSubst (perm, c)
```


(You have defined the `invSubst` function in exercise 2.14.) The `substFwd` function simply returns the character that the given permutation, whether from the plugboard, a rotor, or the reflector. Conversely, `substBwd` returns the character that the given permutation maps *from*, i.e., the character that will be mapped to `c` using the permutation.

Exercise 3.7. Using Cryptol, verify that `substFwd` and `substBwd` return the same elements for each letter in the alphabet for `rotor1`.

Exercise 3.8. Show that `substFwd` and `substBwd` are exactly the same operations for the reflector. Why?

The route back One crucial part of the Enigma is the running of the rotors in reverse after the reflector. Note that this operation ignores the notches, i.e., the rotors do not turn while the signal is passing the second time through the rotors. (In a sense, the rotations happen after the signal completes its full loop, getting to the reflector and back.) Consequently, it is much easer to code as well (compare this code to `joinRotors`, defined in section 3.3):

```
backSignal : {n} (fin n) => ([n]Rotor, Char) -> Char
backSignal (rotors, inputChar) = cs ! 0
  where
    cs = [inputChar] # [  substBwd ([ p | (p, _) <- r ], c)
                        | r <- reverse rotors
                        | c <- cs
                        ]
```

Note that we explicitly reverse the rotors in the definition of `cs`. (The definition of `cs` is another typical example of a fold. See page 20.)

Given all this machinery, coding the entire Enigma loop is fairly straightforward:

```
//enigmaLoop : {n} (fin n) => (Plugboard, [n]Rotor, Reflector, Char)
//                         -> ([n]Rotor, Char)
enigmaLoop (pboard, rotors, refl, c0) = (rotors', c5)
  where
    // 1. First run through the plugboard
    c1 = substFwd (pboard, c0)
    // 2. Now run all the rotors forward
    (rotors', c2) = joinRotors (rotors, c1)
    // 3. Pass through the reflector
    c3 = substFwd (refl, c2)
    // 4. Run the rotors backward
    c4 = backSignal(rotors, c3)
    // 5. Finally, back through the plugboard
    c5 = substBwd (pboard, c4)
```

3.6 The state of the machine

We are almost ready to construct our own Enigma machine in Cryptol. Before doing so, we will take a moment to represent the state of the Enigma machine as a Cryptol record, which will simplify our final construction. At any stage, the state of an Enigma machine is given by the status of its rotors. We will use the following record to represent this state, for an Enigma machine containing `n` rotors:

```
type Enigma n = { plugboard : Plugboard,
                  rotors    : [n]Rotor,
                  reflector : Reflector
                }
```

To initialize an Enigma machine, the operator provides the plugboard settings, rotors, the reflector. Furthermore, the operator also gives the initial positions for the rotors. Rotors can be initially rotated to any position before put together into the machine. We can capture this operation with the function `mkEnigma`:

```
mkEnigma : {n} (Plugboard, [n]Rotor, Reflector, [n]Char)
              -> Enigma n
mkEnigma (pboard, rs, refl, startingPositions) =
    { plugboard  = pboard,
      rotors     = [ r <<< (s - 'A')
                   | r <- rs
                   | s <- startingPositions
                   ],
      reflector  = refl
    }
```

Note how we rotate each given rotor to the left by the amount given by its starting position.

Given this definition, let us construct an Enigma machine out of the components we have created so far, using the starting positions `GCR` for the rotors respectively:

```
modelEnigma : Enigma 3
modelEnigma = mkEnigma (plugboard, [rotor1, rotor2, rotor3],
                        reflector, "GCR")
```

We now have an operational Enigma machine coded up in Cryptol!

3.7 Encryption and decryption

Equipped with all the machinery we now have, coding Enigma encryption is fairly straightforward:

```
enigma : {n, m} (fin n, fin m) => (Enigma n, String m) -> String m
enigma (m, pt) = tail [ c | (_, c) <- rcs ]
  where rcs = [(m.rotors, '*')] #
              [ enigmaLoop (m.plugboard, r, m.reflector, c)
              | c      <- pt
              | (r, _) <- rcs
              ]
```

The function `enigma` takes a machine with `n` rotors and a plaintext of `m` characters, returning a ciphertext of `m` characters back. It is yet another application of the fold pattern, where we start with the initial set of rotors and the placeholder character `*` (which could be anything) to seed the fold. Note how the change in rotors is reflected in each iteration of the fold, through the `enigmaLoop` function. At the end, we simply drop the rotors from `rcs`, and take the `tail` to skip over the seed character `*`.

Here is our Enigma in operation:

```
Cryptol> :set ascii=on
Cryptol> enigma (modelEnigma, "ENIGMAWASAREALLYCOOLMACHINE")
"UPEKTBSDROBVTUJGNCEHHGBXGTF"
```

Decryption As we mentioned before, Enigma was a self-decrypting machine, that is, encryption and decryption are precisely the same operations. Thus, we can define:

```
dEnigma : {n, m} (fin n, fin m) => (Enigma n, String m) -> String m
dEnigma = enigma
```

And decrypt our above message back:

```
Cryptol> dEnigma (modelEnigma, "UPEKTBSDROBVTUJGNCEHHGBXGTF")
"ENIGMAWASAREALLYCOOLMACHINE"
```


We have successfully performed our first Enigma encryption!

Exercise 3.9. Different models of Enigma came with different sets of rotors. You can find various rotor configurations on the web [17]. Create models of these rotors in Cryptol, and run sample encryptions through them.

Exercise 3.10. As we have mentioned before, Enigma implements a polyalphabetic substitution cipher, where the same letter gets mapped to different letters during encryption. The period of a cipher is the number of characters before the encryption repeats itself, mapping the same sequence of letters in the plaintext to the to the same sequence of letters in the ciphertext. What is the period of an Enigma machine with n rotors?

Exercise 3.11. Construct a string of the form `CRYPTOLXXX...XCRYPTOL`, where ...'s are filled with enough number of `X`'s such that encrypting it with our `modelEnigma` machine will map the instances of "`CRYPTOL`" to the same ciphertext. How many `X`'s do you need? What is the corresponding ciphertext for "`CRYPTOL`" in this encryption?

The code You can see all the Cryptol code for our Enigma simulator in Appendix C.

Chapter 4

High-assurance programming

Writing correct software is the holy grail of programming. Bugs inevitably exist, however, even in thoroughly tested projects. One fundamental issue is the lack of support in typical programming languages to let the programmer *state* what it means to be correct, let alone formally establish any notion of correctness. To address this shortcoming, Cryptol advocates the high-assurance programming approach: programmers explicitly state correctness properties along with their code, which are explicitly checked by the Cryptol toolset. Properties are not comments or mere annotations, so there is no concern that they will become obsolete as your code evolves. The goal of this chapter is to introduce you to these tools, and to the notion of high-assurance programming in Cryptol via examples.

4.1 Writing properties

Consider the equality:

$$x^2 - y^2 = (x - y) * (x + y)$$

Let us write two Cryptol functions that capture both sides of this equation:

```
sqDiff1 (x, y) = x^^2  - y^^2
sqDiff2 (x, y) = (x-y) * (x+y)
```

We would like to express the property that `sqDiff1` and `sqDiff2` are precisely the same functions: Given the same `x` and `y`, they should return exactly the same answer. We can express this property in Cryptol using a `properties` declaration:

```
sqDiffsCorrect : ([8], [8]) -> Bit
property sqDiffsCorrect (x, y) = sqDiff1 (x, y) == sqDiff2 (x, y)
```

The above declaration reads as follows: `sqDiffsCorrect` is a property stating that for all `x` and `y`, the expression `sqDiff1 (x, y) == sqDiff2 (x, y)` evaluates to `True`. Furthermore, the type signature restricts the type of the property to apply to only 8-bit values. As usual, the type signature is optional. If not given, Cryptol will infer one for you. Finally, note that the same property can also be expressed (more concisely) using the `===` operator:

```
sqDiffsCorrect : ([8], [8]) -> Bit
property sqDiffsCorrect = sqDiff1 === sqDiff2
```

Note: It is important to emphasize that the mathematical equality above and the Cryptol property are *not* stating precisely the same property. Remember that Cryptol arithmetic depends on the types of the arguments and arithmetic on `[8]` is modular, while the mathematical equation is over arbitrary numbers, including negative, real, or even complex numbers. The takeaway of this discussion is that we are only using this example for illustration purposes: Cryptol properties relate to Cryptol programs, and should not be used for expressing mathematical theorems (unless, of course, you are stating group theory theorems or theorems in an appropriate algebra)! In particular, `sqDiffsCorrect` is a property about the Cryptol functions `sqDiff1` and `sqDiff2`, not about the mathematical equation that inspired it.

Exercise 4.1. Write a property `revRev` stating that `reverse` of a `reverse` returns a sequence unchanged.

Exercise 4.2. Write a property `appAssoc` stating that append is an associative operator.

Exercise 4.3. Write a property `revApp` stating that appending two sequences and reversing the result is the same as reversing the sequences and appending them in the reverse order, as illustrated in the following expression:

```
reverse ("HELLO" # "WORLD") == reverse "WORLD" # reverse "HELLO"
```

Exercise 4.4. Write a property `lshMul` stating that shifting left by k is the same as multiplying by 2^k.

Note: A `property` declaration simply introduces a property about your program, which may or may *not* actually hold. It is an assertion about your program, without any claim of correctness. In particular, you can clearly write properties that simply do not hold:

```
property bogus x = x != x
```

It is important to distinguish between *stating* a property and actually *proving* it. So far, our focus is purely on specification. We will focus on actual proofs in section 4.2.

4.1.1 Property–function correspondence

In Cryptol, properties can be used just like ordinary definitions:

```
Cryptol> sqDiffsCorrect (3, 5)
True
Cryptol> :t sqDiffsCorrect
sqDiffsCorrect : ([8],[8]) -> Bit
```

That is, a property over (x, y) is the same as a function over the tuple `(x, y)`. We call this the property–function correspondence. Property declarations, aside from the slightly different syntax, are *precisely* the same as Cryptol functions whose return type is `Bit`. There is no separate language for writing or working with properties. We simply use the full Cryptol language write both the programs and the properties that they satisfy.

4.1.2 Capturing test vectors

One nice application of Cryptol properties is in capturing test vectors:

```
property inctest = [ f x == y | (x, y) <- testVector ] == ~zero
  where  f x = x + 1
         testVector = [(3, 4), (4, 5), (12, 13), (78, 79)]
```

Notice that the property `inctest` does not have any parameters (no *forall* section), and thus acts as a simple `Bit` value that will be true precisely when the given test case succeeds.

4.1.3 Polymorphic properties

Just like functions, Cryptol properties can be polymorphic as well. If you want to write a property for a polymorphic function, for instance, your properties will naturally be polymorphic too. Here is a simple trivial example:

```
property multShift x = x * 2 == x << 1
```

If we ask Cryptol the type of `multShift`, we get:

```
Cryptol> :t multShift
multShift : {n} (n >= 2, fin n) => [n] -> Bit
```

That is, it is a property about all words of size at least two. The question is whether this property does indeed hold? In the particular case of `multShift` that is indeed the case, below are some examples using the property–function correspondence:


```
Cryptol> multShift (5 : [8])
True
Cryptol> multShift (5 : [10])
True
Cryptol> multShift (5 : [16])
True
```

However, this is *not* always the case for all polymorphic Cryptol properties! The following example demonstrates:

```
property flipNeverIdentity x = x != ~x
```

The property `flipNeverIdentity` states that complementing the bits of a value will always result in a different value: a property we might expect to hold intuitively. Here is the type of `flipNeverIdentity`:

```
Cryptol> :t flipNeverIdentity
flipNeverIdentity : {a} (Eq a, Logic a) => a -> Bit
```

So, the only requirement on `flipNeverIdentity` is that it receives some type that supports comparisons and logical operations. Let us try some examples:

```
Cryptol> flipNeverIdentity True
True
Cryptol> flipNeverIdentity 3
True
Cryptol> flipNeverIdentity [1, 2]
True
```

However:

```
Cryptol> flipNeverIdentity (0 : [0])
False
```

That is, when given a 0-bit word, the complement will in fact do nothing and return its argument unchanged! Therefore, the property `flipNeverIdentity` is not valid, since it holds at certain monomorphic types, but not at all types.

Exercise 4.5. Demonstrate another monomorphic type where `flipNeverIdentity` does *not* hold.

Note: The moral of this discussion is that the notion of polymorphic validity (i.e., that a given polymorphic property will either hold at all of its monomorphic instances or none) does not hold in Cryptol. A polymorphic property can be valid at some, all, or no instances of it.

Exercise 4.6. The previous exercise might lead you to think that it is the 0-bit word type (`[0]`) that is at the root of the polymorphic validity issue. This is not true. Consider the following example:

```
property widthPoly x = (w == 15) || (w == 531)
   where w = length x
```

What is the type of `widthPoly`? At what instances does it hold? Write a property `evenWidth` that holds only at even-width word instances.

4.2 Establishing correctness

Our focus so far has been using Cryptol to *state* properties of our programs, without actually trying to prove them correct. This separation of concerns is essential for a pragmatic development approach. Properties act as contracts that programmers state along with their code, which can be separately checked by the toolset [6]. This approach allows you to state the properties you want, and then work on your code until the properties are indeed satisfied. Furthermore, properties stay with your program forever, so they can be checked at a later time to ensure changes (improvements/additions/optimizations etc.) did not violate the stated properties.

4.2.1 Formal proofs

Recall our very first property, `sqDiffsCorrect`, from section 4.1. We will now use Cryptol to actually prove it automatically. To prove `sqDiffsCorrect`, use the command `:prove`:

```
Cryptol> :prove sqDiffsCorrect
Q.E.D.
```

Note that the above might take a while to complete, as a formal proof is being produced behind the scenes. Once Cryptol formally establishes the property holds, it prints "`Q.E.D.`" to tell the user the proof is complete.

Note: Cryptol uses off-the-shelf SAT and SMT solvers to perform these formal proofs [6]. By default, Cryptol will use Microsoft Research's Z3 SMT solver under the hood, but it can be configured to use other SAT/SMT solvers as well, such as SRI's Yices [24], or CVC4 [15].[1] Note that the `:prove` command is a push-button tool: once the proof starts there is no user involvement. Of course, the external tool used may not be able to complete all the proofs in a reasonable amount of time.

4.2.2 Counterexamples

Of course, properties can very well be invalid, due to bugs in code or the specifications themselves. In these cases, Cryptol will always print a counterexample value demonstrating why the property does not hold. Here is an example demonstrating what happens when things go wrong:

```
failure : [8] -> Bit
property failure x = x == x+1
```

We have:

```
Cryptol> :prove failure
failure 0 = False
```

Cryptol tells us that the property is falsifiable, and then demonstrates a particular value (`0` in this case) that it fails at. These counterexamples are extremely valuable for debugging purposes.

If you try to prove an invalid property that encodes a test vector (subsection 4.1.2), then you will get a mere indication that you have a contradiction, since there are no universally quantified variables to instantiate to show you a counterexample. If the expression evaluates to `True`, then it will be a trivial proof, as expected:

```
Cryptol> :prove False
False = False
Cryptol> :prove True
Q.E.D.
Cryptol> :prove 2 == 3
(2 == 3) = False
Cryptol> :prove reverse [1, 2] == [1, 2]
(reverse [1, 2] == [1,2]) = False
Cryptol> :prove 1+1 == 0
Q.E.D.
```

The very last example demonstrates modular arithmetic in operation, as usual.

4.2.3 Dealing with polymorphism

As we mentioned before, Cryptol properties can be polymorphic. As we explored before, we cannot directly prove polymorphic properties as they may hold for certain monomorphic instances while not for others. In this cases, we must tell Cryptol what particular monomorphic instance of the property we would like it to prove. Let us demonstrate this with the `multShift` property from subsection 4.1.3:

[1] To do this, first install the package(s) from the URLs provided in the bibliography. Once a prover has been installed you can activate it with, for example, `:set prover=cvc4`.

```
Cryptol> :prove multShift
Not a monomorphic type:
{n} (n >= 2, fin n) => [n] -> Bit
```

Cryptol is telling us that it cannot prove a polymorphic property directly. We can, however, give a type annotation to specialize it, and then prove it at a desired instance:

```
Cryptol> :prove multShift : [16] -> Bit
Q.E.D.
```

In fact, you can use this very same technique to pass any bit-valued function to the `:prove` command:

```
Cryptol> :prove dbl where dbl x = (x:[8]) * 2 == x+x
Q.E.D.
```

Of course, a λ-expression (subsection 1.15.3) would work just as well too:

```
Cryptol> :prove \x -> (x:[8]) * 2 == x+x
Q.E.D.
```

Exercise 4.7. Prove the property `revRev` you wrote in exercise 4.1. Try different monomorphic instantiations.

Exercise 4.8. Prove the property `appAssoc` you wrote in exercise 4.2, at several different monomorphic instances.

Exercise 4.9. Prove the property `revApp` you wrote in exercise 4.3, at several different monomorphic instances.

Exercise 4.10. Prove the property `lshMul` you wrote in exercise 4.4, at several different monomorphic instances.

Exercise 4.11. Use the `:prove` command to prove and demonstrate counterexamples for the property `widthPoly` defined in exercise 4.6, using appropriate monomorphic instances.

4.2.4 Conditional proofs

It is often the case that we are interested in a property that only holds under certain conditions. For instance, in exercise 1.37 we have explored the relationship between Cryptol's division, multiplication, and modulus operators, where we asserted the following property:

$$x = (x/y) \times y + (x \,\%\, y)$$

Obviously, this relationship holds only when $y \neq 0$. The idea behind a conditional Cryptol property is that we would like to capture these side-conditions formally in our specifications.

We can use ordinary `if-then-else` expressions in Cryptol to write conditional properties. If the condition is invalid, we simply return `True`, indicating that we are not interested in that particular case. Depending on how natural it is to express the side-condition or its negation, you can use one of the following two patterns:

```
if   side-condition-holds          if    side-condition-fails
then property-expression           then  True
else True                          else  property-expression
```

Alternatively, Cryptol provides a few logical connectives (`==>`, `\/`, and `/\`) that are useful for writing conditional properties. Each is defined in terms of `if-then-else` as follows:

```
a ==> b  =  if a then b else True
a \/ b   =  if a then True else b
a /\ b   =  if a then b else False
```

Exercise 4.12. Express the relationship between division, multiplication, and modulus using a conditional Cryptol property. Prove the property for various monomorphic instances.

Recognizing messages Our work on classic ciphers (chapter 2) and the enigma (chapter 3) involved working with messages that contained the letters `'A'` .. `'Z'` only. When writing properties about these ciphers it will be handy to have a recognizer for such messages, as we explore in the next exercise.

Exercise 4.13. Write a function:

```
validMessage : {n} (fin n) => String n -> Bit
```

that returns `True` exactly when the input only consists of the letters `'A'` through `'Z'`. (**Hint** Use the functions `all` defined in exercise 1.54, and `elem` defined in exercise 1.61.)

Exercise 4.14. Recall the pair of functions `caesar` and `dCaesar` from section 2.1. Write a property, named `caesarCorrect`, stating that `caesar` and `dCaesar` are inverses of each other for all `d` (shift amount) and `msg` (message). Is your property valid? What extra condition do you need to assert on `msg` for your property to hold? Prove the property for all messages of length 10.

Exercise 4.15. Write and prove a property for the `modelEnigma` machine (page 38), relating the `enigma` and `dEnigma` functions from section 3.7.

This may take a long time to prove, depending on the speed of your machine, and the prover you choose.

4.3 Automated random testing

Cryptol's `:prove` command constructs rigorous formal proofs using push-button tools.[2] The underlying technique used by Cryptol (SAT- and SMT-based equivalence checking) is complete; i.e., it will always either prove the property or find a counterexample. In the worst case, however, the proof process might take infeasible amounts of resources, potentially running out of memory or taking longer than the amount of time you are willing to wait.

What is needed for daily development tasks is a mechanism to gain some confidence on the correctness of the properties without paying the price of formally proving them. This is the goal of Cryptol's `:check` command, inspired by Haskell's quick-check library [3]. Instead of trying to formally prove your property, `:check` tests it at random values to give you quick feedback. This approach is very suitable for rapid development. By using automated testing you get frequent and quick updates from Cryptol regarding the status of your properties, as you work through your code. If you introduce a bug, it is likely (although not guaranteed) that the `:check` command will alert you right away. Once you are satisfied with your code, you can use the `:prove` command to conduct the formal proofs, potentially leaving them running overnight.

The syntax of the `:check` command is precisely the same as the `:prove` command. By default, it will run your property over 100 randomly generated test cases.

Exercise 4.16. Use the `:check` command to test the property `caesarCorrect` you have defined in exercise 4.13, for messages of length 10. Use the command `:set tests=1000` to change the number of test cases to 1,000. Observe the test coverage statistics reported by Cryptol. How is the total number of cases computed?

Exercise 4.17. If the property is *small* in size, `:check` might as well prove/disprove it. Try the following commands:

```
:check True
:check False
:check \x -> x==(x:[8])
```

Exercise 4.18. Write a bogus property that will be very easy to disprove using `:prove`, while `:check` will have a hard time obtaining the counterexample. The moral of this exercise is that you should try `:prove` early in your development and not get too comfortable with the results of `:check`!

Exhaustive testing The `:exhaust` command is a variant of `:check` that always uses exhaustive testing: It tests the property on every possible combination of inputs. The number of tests run by `:check` is limited to the value of `:set`

[2]While some of the solvers that Cryptol uses are capable of *emitting* proofs, such functionality is not exposed as a Cryptol feature.

`tests`, but `:exhaust` may run a much larger number. If the argument types are not sufficiently small, `:exhaust` may run for longer than you are willing to wait!

Exercise 4.19. Try the following command to check a conjecture about arithmetic.

```
:check \(x:[16]) -> (x + 6) / 3 == x / 3 + 2
```

What is the result? Does the response give you much confidence in the property? Now try checking the same predicate using `:exhaust`. Does the property hold in general?

Bulk operations If you use `:check` and `:prove` commands without any arguments, Cryptol will check and prove all the properties defined in your program. This is a simple means of exercising all your properties automatically.

4.4 Checking satisfiability

Closely related to proving properties is the notion of checking satisfiability. In satisfiability checking, we would like to find arguments to a bit-valued function such that it will evaluate to `True`, i.e., it will be satisfied.

One way to think about satisfiability checking is *intelligently* searching for a solution. Here is a simple example. Let us assume we would like to compute the modular square root of 9 as an 8-bit value. The obvious solution is `3`, of course, but we are wondering if there are other solutions to the equation $x^2 \equiv 9 \pmod{2^8}$. To get started, let us first define a function that will return `True` if its argument is a square root of 9:

```
isSqrtOf9 : [8] -> Bit
isSqrtOf9 x = x*x == 9
```

Any square root of 9 will make the function `isSqrtOf9` return `True`, i.e., it will *satisfy* it. Thus, we can use Cryptol's satisfiability checker to find those values of `x` automatically:

```
Cryptol> :sat isSqrtOf9
isSqrtOf9 3 = True
```

Not surprisingly, Cryptol told us that 3 is one such value. We can search for other solutions by explicitly disallowing 3:

```
Cryptol> :sat \x -> isSqrtOf9 x && ~(elem x [3])
\x -> isSqrtOf9 x && ~(elem (x, [3])) 131 = True
```

Note the use of the λ-expression to indicate the new constraint. (Of course, we could have defined another function `isSqrtOf9ButNot3` for the same effect, but the λ-expression is really handy in this case.) We have used the function `elem` you have defined in exercise 1.61 to express the constraint `x` must not be 3. In response, Cryptol told us that `125` is another solution. Indeed $125 * 125 = 9 \pmod{2^7}$, as you can verify separately. We can search for more:

```
Cryptol> :sat \x -> isSqrtOf9 x && ~(elem x [3, 125])
\x -> isSqrtOf9 x && ~(elem (x, [3, 131])) 253 = True
```

Rather than manually adding solutions we have already seen, we can search for other solutions by asking the satisfiability checker for more solutions using the `satNum` setting:

```
Cryptol> :set satNum = 4
Cryptol> :sat isSqrtOf9
isSqrtOf9 3 = True
isSqrtOf9 131 = True
isSqrtOf9 125 = True
isSqrtOf9 253 = True
```

By default, `satNum` is set to `1`, so we only see one solution. When we change it to `4`, the satisfiability checker will try to find *up to* 4 solutions. We can also set it to `all`, which will try to find as many solutions as possible.

```
Cryptol> :set satNum = all
Cryptol> :sat isSqrtOf9
isSqrtOf9 3 = True
isSqrtOf9 131 = True
isSqrtOf9 125 = True
isSqrtOf9 253 = True
```

So, we can rest assured that there are exactly four 8-bit square roots of 9: namely 3, 131, 125, and 253. (Note that Cryptol can return the satisfying solutions in any order depending on the backend-solver and other configurations. What is guaranteed is that you will get precisely the same set of solutions at the end. Also be aware that, especially when using the `Integer` type, the number of solutions may be very large or even infinite.)

The whole point of the satisfiability checker is to be able to quickly search for particular values that are solutions to potentially complicated bit-valued functions. In this sense, satisfiability checking can also be considered as an automated way to invert a certain class of functions, going back from results to arguments. Of course, this search is not done blindly, but rather using SAT and SMT solvers to quickly find the satisfying values. Cryptol's `:sat` command hides the complexity, allowing the user to focus on the specification of the problem.

Exercise 4.20. Fermat's last theorem states that there are no integer solutions to the equation $a^n + b^n = c^n$ when $a, b, c > 0$, and $n > 2$. While we can code Fermat's theorem in Cryptol using the `Integer` type, the SMT solver backends used by Cryptol do not support symbolic exponentiation on unbounded integers. However, we can make more progress with the modular version of it where the exponentiation and addition are done modulo a fixed bit-size. Write a function `modFermat` with the following signature:

```
type Quad a = ([a], [a], [a], [a])
modFermat : {s} (fin s, s >= 2) => Quad s -> Bit
```

such that `modFermat (a, b, c, n)` will return `True` if the modular version of Fermat's equation is satisfied by the values of `a`, `b`, `c`, and `n`. Can you explain why you need the constraints `fin s` and `s >= 2`?

Exercise 4.21. Use the `:sat` command to see if there are any satisfying values for the modular version of Fermat's last theorem for various bit sizes. Surprised? What can you conclude from your observations?

Chapter 5

AES: The Advanced Encryption Standard

AES is a symmetric key encryption algorithm (a symmetric cipher, per the discussion in chapter 2), based on the Rijndael algorithm designed by Joan Daemen and Vincent Rijmen [4]. (The term *symmetric key* means that the algorithm uses the same key for encryption and decryption.) AES was adopted in 2001 by the US government, deemed suitable for protecting classified information up to *secret* level for the key size 128, and up to the *top-secret* level for key sizes 192 and 256.

In this chapter, we will program AES in Cryptol. Our emphasis will be on clarity, as opposed to efficiency, and we shall follow the NIST standard description of AES fairly closely [12]. Referring to the standard as you work your way through this chapter is recommended.

Some surprises may be at hand for the reader who has never deeply examined a modern cryptography algorithm before.

First, algorithms like AES are typically composed of many smaller units of varying kinds. Consequently, the entire algorithm is constructed bottom-up by specifying and verifying each of its component pieces. It is wise to handle smaller and simpler components first. It is also a good practice, though hard to accomplish the first one or two times you write such a specification, to write specifications with an eye toward reuse in multiple instantiations of the same algorithm (e.g., different key sizes or configurations). The choice between encoding configurations at the type level or the value level is aesthetic and practical: some verification is only possible when one encodes information at the type level.

Second, algorithms frequently depend upon interesting data structures and mathematical constructs, the latter of which can be though of as data structures in a pure mathematics sense. The definition, shape, form, and subtleties of these data structures are critical to the *correct definition* of the crypto algorithm *as well as its security properties*. Implementing an algorithm using an alternative datatype construction that you believe has the same properties as that which is stipulated in a standard is nearly always the wrong thing to do. Also, the subtleties of these constructions usually boils down to what an engineer might think of as "magic numbers"—strange initial values or specific polynomials that appear out of thin air. Just remind yourself that the discovery and analysis of those magic values was, in general, the joint hard work of a community of cryptographers.

5.1 Parameters

The AES algorithm always takes 128 bits of input, and always produces 128 bits of output, regardless of the key size. The key-size can be one of 128 (AES128), 192 (AES192), or 256 (AES256). Following the standard, we define the following three parameters [12, section 2.2]:

- `Nb`: Number of columns, always set to 4 by the standard.
- `Nk`: Number of key-blocks, which is the number of 32-bit words in the key: 4 for AES128, 6 for AES192, and 8 for AES256;
- `Nr`: Number of rounds, which `Nr` is always $6 + \texttt{Nk}$, according to the standard. Thus, 10 for AES128, 12 for AES192, and 14 for AES256.

The Cryptol definitions follow the above descriptions verbatim:

```
type AES128 = 4
type AES192 = 6
type AES256 = 8

type Nk = AES128
type Nb = 4
type Nr = 6 + Nk
```

The following derived type is helpful in signatures:

```
type AESKeySize = (Nk*32)
```

5.2 Polynomials in GF(2^8)

AES works on a two-dimensional representation of the input arranged into bytes, called the *state*. For a 128-bit input, we have precisely 4 rows, each containing `Nb` (i.e., 4) bytes, each of which is 8 bits wide, totaling $4 \times 4 \times 8 = 128$ bits. The bytes themselves are treated as finite field elements in the Galois field GF(2^8) [19], giving rise to the following declarations:

```
type GF28  = [8]
type State = [4][Nb]GF28
```

The hard-encoding of `GF28` in this specification is completely appropriate because the construction of AES depends entirely upon the Galois field GF(2^8). It is conceivable that other algorithms might be parameterized across GF(2^k) for some k, in which case the underlying type declaration would also be parameterized.

While a basic understanding Galois field operations is helpful, the details are not essential for our modeling purposes. It suffices to note that GF(2^8) has precisely 256 elements, each of which is a polynomial of maximum degree 7, where the coefficients are either 0 or 1. The numbers from 0 to 255 (i.e., all possible 8-bit values) are in one-to-one correspondence with these polynomials. The coefficients of the polynomial come from the successive bits of the number, and vice versa. For instance the 8-bit number 87 can be written as `0b01010111` in binary, and hence corresponds to the polynomial $x^6 + x^4 + x^2 + x^1 + 1$. Similarly, the polynomial $x^4 + x^3 + x$ corresponds to the number `0b00011010`, i.e., 26. We can also compute this value by evaluating the polynomial for $x = 2$, obtaining $2^4 + 2^3 + 2 = 16 + 8 + 2 = 26$.

Cryptol allows you to write polynomials in GF(2^n), for arbitrary n, using the following notation:

```
Cryptol> <| x^^6 + x^^4 + x^^2 + x^^1 + 1 |>
87
Cryptol> 0b1010111
87
Cryptol> <| x^^4 + x^^3 + x |>
26
Cryptol> 0b11010
26
```

A polynomial is similar to a decimal representation of a number, albeit in a more suggestive syntax. Like with a decimal, the Cryptol type system will default the type to be the smallest number of bits required to represent the polynomial, but it may be expanded to more bits if an expression requires it.

Addition and Subtraction Given two polynomials, adding and subtracting them in a Galois field GF(2^n) results in a new polynomial where terms with the same power cancel each other out. When interpreted as a word, both addition and subtraction amount to a simple exclusive-or operation. Cryptol's `^` operator captures this idiom concisely:

```
Cryptol> <| x^^4 + x^^2 |> ^ <| x^^5 + x^^2 + 1 |> == \
         <| x^^5 + x^^4 + 1 |>
True
```

Note that the term x^2 cancels since it appears in both polynomials.

Exercise 5.1. Adding a polynomial to itself in GF(2^n) will always yield `0` since all the terms will cancel each other. Write and prove a theorem `polySelfAdd` over `GF28` to state this fact.

While adding two polynomials does not warrant a separate function, we will need a version that can add a sequence of polynomials:

Exercise 5.2. Define a function

```
gf28Add : {n} (fin n) => [n]GF28 -> GF28
```

that adds all the elements given. (**Hint** Use a fold, see page 20.)

Multiplication Multiplication in $GF(2^n)$ follows the usual polynomial multiplication algorithm, where we multiply the first polynomial with each term of the second, and add all the partial sums (i.e., compute their exclusive-or). While this operation can be programmed explicitly, Cryptol does provide the primitive `pmult` for this purpose:

```
Cryptol> pmult <| x^^3 + x^^2 + x + 1 |> <| x^^2 + x + 1 |>
45
Cryptol> <| x^^5 + x^^3 + x^^2 + 1 |>
45
```

Exercise 5.3. Multiply the polynomials $x^3 + x^2 + x + 1$ and $x^2 + x + 1$ by hand in $GF(2^8)$ and show that the result is indeed $x^5 + x^3 + x^2 + 1$, (i.e., 45), justifying Cryptol's result above.

Reduction If you multiply two polynomials with degrees m and n, you will get a new polynomial of degree $m + n$. As we mentioned before, the polynomials in $GF(2^8)$ have degree at most 7. Obviously, $m + n$ can be larger than 7 when we multiply to elements of $GF(2^8)$. So we have to find a way to map the resulting larger-degree polynomial back to an element of $GF(2^8)$. This is done by reduction, or modulus, with respect to an *irreducible polynomial*. The AES algorithm uses the following polynomial for this purpose:

```
irreducible = <| x^^8 + x^^4 + x^^3 + x + 1 |>
```

(Recall in the introduction of this chapter our warning about magic!)

Note that `irreducible` is *not* an element of $GF(2^8)$, since it has degree 8. However we can use this polynomial to define the multiplication routine itself, which uses Cryptol's `pmod` (polynomial modulus) function, as follows:

```
gf28Mult : (GF28, GF28) -> GF28
gf28Mult (x, y) = pmod (pmult x y) irreducible
```

Polynomial modulus and division operations follow the usual schoolbook algorithm for long-division—a fairly laborious process in itself, but it is well studied in mathematics [20]. Luckily for us, Cryptol's `pdiv` and `pmod` functions implement these operations, saving us the programming task.

Exercise 5.4. Divide $x^5 + x^3 + 1$ by $x^3 + x^2 + 1$ by hand, finding the quotient and the remainder. Check your answer with Cryptol's `pmod` and `pdiv` functions.

Exercise 5.5. Write and prove theorems showing that `gf28Mult` (i) has the polynomial 1 as its unit, (ii) is commutative, and (iii) is associative.

5.3 The `SubBytes` transformation

Recall that the state in AES is a 4×4 matrix of bytes. As part of its operation, AES performs the so called `SubBytes` transformation [12, section 5.1.1], substituting each byte in the state with another element. Given an $x \in GF(2^8)$, the substitution for x is computed as follows:

1. Compute the multiplicative inverse of x in $GF(2^8)$, call it b. If x is 0, then take 0 as the result.
2. Replace bits in b as follows: Each bit b_i becomes $b_i \oplus b_{i+4 \pmod 8} \oplus b_{i+5 \pmod 8} \oplus b_{i+6 \pmod 8} \oplus b_{i+7 \pmod 8} \oplus c_i$. Here $\oplus$ is exclusive-or and c is `0x63`.

Computing the multiplicative inverse It turns out that the inverse of any non-zero x in GF(2^8) can be computed by raising x to the power 254, i.e., multiplying x by itself 254 times. (Mathematically, GF(2^8) is a cyclic group such that x^{255} is always 1 for any x, making x^{254} the multiplicative inverse of x.)

Exercise 5.6. Write a function

```
gf28Pow : (GF28, [8]) -> GF28
```

such that the call `gf28Pow (n, k)` returns n^k using `gf28Mult` as the multiplication operator. (**Hint** Use the fact that $x^0 = 1$, $x^{2n} = (x^n)^2$, and $x^{2n+1} = x \times (x^n)^2$ to speed up the exponentiation.)

Exercise 5.7. Write a function

```
gf28Inverse : GF28 -> GF28
```

to compute the multiplicative inverse of a given element by raising it to the power `254`. Note that `gf28Inverse` must map `0` to `0`. Do you have to do anything special to make sure this happens?

Exercise 5.8. Write and prove a property `gf28InverseCorrect`, ensuring that `gf28Inverse x` does indeed return the multiplicative inverse of `x`. Do you have to do anything special when `x` is `0`?

Transforming the result As we mentioned above, the AES specification asks us to transform each bit b_i according to the following transformation:

$$b_i \oplus b_{i+4 \pmod 8} \oplus b_{i+5 \pmod 8} \oplus b_{i+6 \pmod 8} \oplus b_{i+7 \pmod 8} \oplus c_i$$

For instance, bit b_5 becomes $b_5 \oplus b_1 \oplus b_2 \oplus b_3 \oplus c_5$. When interpreted at the word level, this basically amounts to computing:

$$b \oplus (b \ggg 4) \oplus (b \ggg 5) \oplus (b \ggg 6) \oplus (b \ggg 7) \oplus c$$

by aligning the corresponding bits in the word representation.

Exercise 5.9. Write a function

```
xformByte : GF28 -> GF28
```

that computes the above described transformation.

Putting it together Armed with `gf28Inverse` and `xformByte`, we can easily code the function that transforms a single byte as follows:

```
SubByte : GF28 -> GF28
SubByte b = xformByte (gf28Inverse b)
```

AES's `SubBytes` transformation merely applies this function to each byte of the state:

```
SubBytes : State -> State
SubBytes state = [ [ SubByte b | b <- row ] | row <- state ]
```

Table lookup Our definition of the `SubByte` function above follows how the designers of AES came up with the substitution maps, i.e., it is a *reference* implementation. For efficiency purposes, however, we might prefer a version that simply performs a look-up in a table. Notice that the type of `SubByte` is `GF28 -> GF28`, i.e., it takes a value between 0 and 255. Therefore, we can make a table containing the precomputed values for all possible 256 inputs, and simply perform a table look-up instead of computing these values each time we need it. In fact, Figure 7 on page 16 of the AES standard lists these precomputed values for us [12, section 5.1.1]. We capture this table below in Cryptol:


```
sbox : [256]GF28
sbox = [
   0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67,
   0x2b, 0xfe, 0xd7, 0xab, 0x76, 0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59,
   0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, 0xb7,
   0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1,
   0x71, 0xd8, 0x31, 0x15, 0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05,
   0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, 0x09, 0x83,
   0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29,
   0xe3, 0x2f, 0x84, 0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b,
   0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, 0xd0, 0xef, 0xaa,
   0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c,
   0x9f, 0xa8, 0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc,
   0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, 0xcd, 0x0c, 0x13, 0xec,
   0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19,
   0x73, 0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee,
   0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, 0xe0, 0x32, 0x3a, 0x0a, 0x49,
   0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
   0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4,
   0xea, 0x65, 0x7a, 0xae, 0x08, 0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6,
   0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, 0x70,
   0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9,
   0x86, 0xc1, 0x1d, 0x9e, 0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e,
   0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, 0x8c, 0xa1,
   0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0,
   0x54, 0xbb, 0x16]
```

With this definition of `sbox`, the look-up variants of `SubByte` and `SubBytes` becomes:

```
SubByte' : GF28 -> GF28
SubByte' x = sbox @ x

SubBytes' : State -> State
SubBytes' state = [ [ SubByte' b | b <- row ] | row <- state ]
```

Exercise 5.10. Write and prove a property stating that `SubByte` and `SubByte'` are equivalent.

Note: The `SubByte'` and `SubBytes'` versions are going to be more efficient for execution, naturally. We should emphasize that this mode of development is quite common in modern cryptography. Ciphers are typically designed using ideas from mathematics, often requiring complicated algorithms. To speed things up, however, implementors use clever optimization tricks, convert functions to look-up tables using precomputed values, etc.
What Cryptol allows us to do is to write the algorithms using both styles, and then formally show that they are indeed equivalent, as you did in exercise 5.10 above. This mode of high-assurance development makes sure that we have not made any cut-and-paste errors when we wrote down the numbers in our `sbox` table. Equivalently, our proof also establishes that the official specification [12, section 5.1.1] got its own table correct!

5.4 The `ShiftRows` transformation

The second transformation AES utilizes is the `ShiftRows` operation [12, section 5.1.2]. This operation treats the `State` as a 4×4 matrix, and rotates the last three row to the left by the amounts 1, 2, and 3, respectively. Implementing `ShiftRows` in Cryptol is trivial, using the `<<<` operator:

```
ShiftRows : State -> State
ShiftRows state = [ row <<< shiftAmount | row <- state
                                        | shiftAmount <- [0 .. 3]
                  ]
```

Exercise 5.11. Can you transform a state back into itself by repeated applications of `ShiftRows`? How many times would you need to shift? Verify your answer by writing and proving a corresponding Cryptol property.

5.5 The `MixColumns` transformation

The third major transformation AES performs is the `MixColumns` operation [12, section 5.1.3]. In this transformation, the `State` is viewed as a 4×4 matrix, and each successive column of it is replaced by the result of multiplying it by the matrix:

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

As you might recall from linear algebra, given two *compatible* matrices A and B, the ijth element of $A \times B$ is the dot-product of the ith row of A and the jth column of B. (By *compatible* we mean the number of columns of A must be the same as the number of rows of B. All our matrices are 4×4, so they are always compatible.) The dot-product is defined as multiplying the corresponding elements of two same-length vectors and adding the results together. The only difference here is that we use the functions `gf28Add` and `gf28Mult` for addition and multiplication respectively. We will develop this algorithm in the following sequence of exercises.

Exercise 5.12. Write a function `gf28DotProduct` with the signature:

```
gf28DotProduct : {n} (fin n) => ([n]GF28, [n]GF28) -> GF28
```

such that `gf28DotProduct` returns the dot-product of two length n vectors of $\mathrm{GF}(2^8)$ elements.

Exercise 5.13. Write properties stating that the vector operation `gf28DotProduct` is commutative and distributive over vector addition:

$$\begin{aligned} a \cdot b &= b \cdot a \\ a \cdot (b + c) &= a \cdot b + a \cdot c \end{aligned}$$

Addition over vectors is defined element-wise. Prove the commutativity property for vectors of length 10. Distributivity will take much longer, so you might want to do a `:check` on it.

Exercise 5.14. Write a function

```
gf28VectorMult : {n, m} (fin n) => ([n]GF28, [m][n]GF28) -> [m]GF28
```

computing the dot-product of its first argument with each of the m rows of the second argument, returning the resulting values as a vector of m elements.

Exercise 5.15. Write a function

```
gf28MatrixMult : {n, m, k} (fin m) => ([n][m]GF28, [m][k]GF28) -> [n][k]GF28
```

which multiplies the given matrices in $\mathrm{GF}(2^8)$.

Now that we have the matrix multiplication machinery built, we can code `MixColumns` fairly easily. Following the description in the AES standard [12, section 5.3.1], all we have to do is to multiply the matrix we have seen at the beginning of this section with the state:

```
MixColumns : State -> State
MixColumns state = gf28MatrixMult (m, state)
  where m = [[2, 3, 1, 1],
             [1, 2, 3, 1],
             [1, 1, 2, 3],
             [3, 1, 1, 2]]
```

Note that Cryptol makes no built-in assumption about row- or column-ordering of multidimensional matrices. Of course, given Cryptol's concrete syntax, it makes little sense to do anything but row-based ordering.

5.6 Key expansion

Recall from section 5.1 that AES takes 128, 192, or 256-bit keys. The key is not used as-is, however. Instead, AES expands the key into a number of round keys, called the *key schedule*. Construction of the key schedule relies on a number of auxiliary definitions, as we shall see shortly.

Round constants The AES standard refers to the constant array `Rcon` used during key expansion. For each `i`, `Rcon[i]` contains 4 words, the last three being 0 [12, section 5.2]. The first element is given by x^{i-1}, where exponentiation is done using the `gf28Pow` function you have defined in exercise 5.6. In Cryptol, it is easiest to define `Rcon` as a function:

```
Rcon : [8] -> [4]GF28
Rcon i = [(gf28Pow (<| x |>, i-1)), 0, 0, 0]
```

Exercise 5.16. By definition, AES only calls `Rcon` with the parameters ranging from 1–10. Based on this, create a table-lookup version

```
Rcon' : [8] -> [4]GF28
```

that simply performs a look-up instead. (**Hint** Use Cryptol to find out what the elements of your table should be.)

Exercise 5.17. Write and prove a property that `Rcon` and `Rcon'` are equivalent when called with numbers in the range 1–10.

The `SubWord` function The AES specification refers to a function named `SubWord` [12, section 5.2], that takes a 32-bit word and applies the `SubByte` transformation from section 5.3. This function is trivial to code in Cryptol:

```
SubWord : [4]GF28 -> [4]GF28
SubWord bs = [ SubByte' b | b <- bs ]
```

Note that we have used the table-lookup version (`SubByte'`, page 53) above.

The `RotWord` function The last function we need for key expansion is named `RotWord` by the AES standard [12, section 5.2]. This function merely rotates a given word cyclically to the left:

```
RotWord : [4]GF28 -> [4]GF28
RotWord [a0, a1, a2, a3] = [a1, a2, a3, a0]
```

We could have used `<<<` to implement `RotWord` as well, but the above definition textually looks exactly the one given in the standard specification, and hence is preferable for the purposes of clarity.

The key schedule Recall that AES operates on 128, 192, or 256 bit keys. These keys are used to construct a sequence of so-called *round keys*, each of which is 128 bits wide, and is viewed the same way as the `State`:

```
type RoundKey = State
```

The expanded key schedule contains `Nr`+1 round-keys. (Recall from section 5.1 that `Nr` is the number of rounds.) It also helps to separate out the first and the last keys, as they are used in a slightly different fashion. Based on this discussion, we use the following Cryptol type to capture the key schedule:

```
type KeySchedule = (RoundKey, [Nr-1]RoundKey, RoundKey)
```

The key schedule is computed by seeding it with the initial key and computing the successive elements from the previous entries. In particular, the ith element of the expanded key is determined as follows, copied verbatim from the AES specification [12, figure 11; section 5.2]:

```
temp = w[i-1]
if (i mod Nk = 0)
    temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
else if (Nk > 6 and i mod Nk = 4)
    temp = SubWord(temp)
end if
w[i] = w[i-Nk] xor temp
```

In the pseudo-code, the `w` array contains the expanded key. We are computing `w[i]`, using the values `w[i-1]` and `w[i-Nk]`. The result is the exclusive-or of `w[i-Nk]` and a mask value, called `temp` above. The mask is computed using `w[i-1]`, the `Rcon` array we have seen before, the current index `i`, and `Nk`. This computation is best expressed as a function in Cryptol that we will call `NextWord`. We will name the `w[i-1]` argument `prev`, and the `w[i-Nk]` argument `old`. Otherwise, the Cryptol code just follows the pseudo-code above, written in a functional style to compute the mask:

```
NextWord : ([8],[4][8],[4][8]) -> [4][8]
NextWord(i, prev, old) = old ^ mask
   where mask = if i % `Nk == 0
                then SubWord(RotWord(prev)) ^ Rcon' (i / `Nk)
                else if (`Nk > 6) && (i % `Nk == 4)
                     then SubWord(prev)
                     else prev
```

Note: It is well worth studying the pseudo-code above and the Cryptol equivalent to convince yourself they are expressing the same idea!

To compute the key schedule we start with the initial key as the seed. We then make calls to `NextWord` with a sliding window of `Nk` elements, computing the subsequent elements. Let us first write a function that will apply this algorithm to generate an infinite regression of elements:

```
ExpandKeyForever : [Nk][4][8] -> [inf]RoundKey
ExpandKeyForever seed = [ transpose g | g <- groupBy`{4} keyWS ]
  where keyWS = seed # [ NextWord(i, prev, old)
                       | i    <- [`Nk ...]
                       | prev <- drop`{Nk-1} keyWS
                       | old  <- keyWS
                       ]
```

Note how `prev` tracks the previous 32 bits of the expanded key (by dropping the first `Nk-1` elements), while `old` tracks the `i-Nk`th recurrence for `keyWS`. Once we have the infinite expansion, it is easy to extract just the amount we need by using number of rounds (`Nr`) as our guide:

```
ExpandKey : [AESKeySize] -> KeySchedule
ExpandKey key = (keys @ 0, keys @@ [1 .. (Nr - 1)], keys @ `Nr)
  where  seed : [Nk][4][8]
         seed = split (split key)
         keys = ExpandKeyForever seed
```

The call `split key` chops `AESKey` into `[Nk*4][8]`, and the outer call to `split` further constructs the `[Nk][4][8]` elements.

Testing `ExpandKey` The completion of `ExpandKey` is an important milestone in our AES development, and it is worth testing it before we proceed. The AES specification has example key expansions that we can use. The following function will be handy in viewing the output correctly aligned:

```
fromKS : KeySchedule -> [Nr+1][4][32]
fromKS (f, ms, l) = [ formKeyWords (transpose k)
                    | k <- [f] # ms # [l]
                    ]
       where formKeyWords bbs = [ join bs | bs <- bbs ]
```

Here is the example from Appendix A.1 of the AES specification [12]:

```
Cryptol> fromKS (ExpandKey 0x2b7e151628aed2a6abf7158809cf4f3c)
[[0x2b7e1516, 0x28aed2a6, 0xabf71588, 0x09cf4f3c],
 [0xa0fafe17, 0x88542cb1, 0x23a33939, 0x2a6c7605],
 [0xf2c295f2, 0x7a96b943, 0x5935807a, 0x7359f67f],
 [0x3d80477d, 0x4716fe3e, 0x1e237e44, 0x6d7a883b],
 [0xef44a541, 0xa8525b7f, 0xb671253b, 0xdb0bad00],
 [0xd4d1c6f8, 0x7c839d87, 0xcaf2b8bc, 0x11f915bc],
 [0x6d88a37a, 0x110b3efd, 0xdbf98641, 0xca0093fd],
 [0x4e54f70e, 0x5f5fc9f3, 0x84a64fb2, 0x4ea6dc4f],
 [0xead27321, 0xb58dbad2, 0x312bf560, 0x7f8d292f],
 [0xac7766f3, 0x19fadc21, 0x28d12941, 0x575c006e],
 [0xd014f9a8, 0xc9ee2589, 0xe13f0cc8, 0xb6630ca6]]
```

As you can verify this output matches the last column of the table in Appendix A.1 of the reference specification for AES.

5.7 The `AddRoundKey` transformation

`AddRoundKey` is the simplest of all the transformations in AES [12, section 5.1.4]. It merely amounts to the exclusive-or of the state and the current round key:

```
AddRoundKey : (RoundKey, State) -> State
AddRoundKey (rk, s) = rk ^ s
```

Notice that Cryptol's `^` operator applies structurally to arbitrary shapes, computing the exclusive-or element-wise.

5.8 AES encryption

We now have all the necessary machinery to perform AES encryption.

AES rounds As mentioned before, AES performs encryption in rounds. Each round consists of performing `SubBytes` (section 5.3), `ShiftRows` (section 5.4), and `MixColumns` (section 5.5). Before finishing up, each round also adds the current round key to the state [12, section 5.1]. The Cryptol code for the rounds is fairly trivial:

```
AESRound : (RoundKey, State) -> State
AESRound (rk, s) = AddRoundKey (rk, MixColumns (ShiftRows (SubBytes s)))
```

The final round The last round of AES is slightly different than the others. It omits the `MixColumns` transformation:

```
AESFinalRound : (RoundKey, State) -> State
AESFinalRound (rk, s) = AddRoundKey (rk, ShiftRows (SubBytes s))
```

Forming the input/output blocks Recall that AES processes input in blocks of 128 bits, producing 128 bits of output, regardless of the key size. We will need two helper functions to convert 128-bit messages to and from AES states. Conversion from a message to a state is easy to define:

```
msgToState : [128] -> State
msgToState msg = transpose (split (split msg))
```

The first call to `split` gives us four 32-bit words, which we again split into bytes. We then form the AES state by transposing the resulting matrix. In the other direction, we simply transpose the state and perform the necessary `join`s:

```
stateToMsg : State -> [128]
stateToMsg st = join (join (transpose st))
```

Exercise 5.18. Write and prove a pair of properties stating that `msgToState` and `stateToMsg` are inverses of each other.

Putting it together To encrypt, AES merely expands the given key and calls the round functions. The starting state (`state0` below) is constructed by adding the first round key to the input. We then run all the middle rounds using a simple comprehension, and finish up by applying the last round [12, figure 5, section 5.1]:

```
aesEncrypt : ([128], [AESKeySize]) -> [128]
aesEncrypt (pt, key) = stateToMsg (AESFinalRound (kFinal, rounds ! 0))
  where   (kInit, ks, kFinal) = ExpandKey key
          state0 = AddRoundKey(kInit, msgToState pt)
          rounds = [state0] # [ AESRound (rk, s) | rk <- ks
                                                 | s <- rounds
                              ]
```

Testing We can now run some test vectors. Note that, just because a handful of test vectors pass, we cannot claim that our implementation of AES is correct.

The first example comes from Appendix B of the AES standard [12]:

```
Cryptol> aesEncrypt (0x3243f6a8885a308d313198a2e0370734,   \
                     0x2b7e151628aed2a6abf7158809cf4f3c)
0x3925841d02dc09fbdc118597196a0b32
```

which is what the standard asserts to be the answer. (Note that you have to read the final box in Appendix B column-wise!) The second example comes from Appendix C.1:

```
Cryptol> aesEncrypt (0x00112233445566778899aabbccddeeff,   \
                     0x000102030405060708090a0b0c0d0e0f)
0x69c4e0d86a7b0430d8cdb78070b4c55a
```

Again, the result agrees with the standard.

Other key sizes Our development of AES has been key-size agnostic, relying on the definition of the parameter `Nk`. (See section 5.1.) To obtain AES192, all we need is to set `Nk` to be 6, no additional code change is needed. Similarly, we merely need to set `Nk` to be 8 for AES256.

Exercise 5.19. By setting `Nk` to be 6 and 8 respectively, try the test vectors given in Appendices C.2 and C.3 of the AES standard [12].

5.9 Decryption

AES decryption is fairly similar to encryption, except it uses inverse transformations [12, figure 12, section 5.3]. Armed with all the machinery we have built so far, the inverse transformations are relatively easy to define.

5.9.1 The `InvSubBytes` transformation

The `InvSubBytes` transformation reverses the `SubBytes` transformation of section 5.3. As with `SubBytes`, we have a choice to either do a table lookup implementation, or follow the mathematical description. We will do the former in these examples; you are welcome to do the latter on your own and prove the equivalence of the two versions. To do so, we need to invert the transformation given by:

$$b \oplus b \ggg 4 \oplus b \ggg 5 \oplus b \ggg 6 \oplus b \ggg 7 \oplus c$$

where c is `0x63`. It turns out that the inverse of this transformation can be computed by

$$b \ggg 2 \oplus b \ggg 5 \oplus b \ggg 7 \oplus d$$

where d is `0x05`. It is easy to code this inverse transform in Cryptol:


```
xformByte' : GF28 -> GF28
xformByte' b = gf28Add [(b >>> 2), (b >>> 5), (b >>> 7), d]
  where d = 0x05
```

Exercise 5.20. Write and prove a Cryptol property stating that `xformByte'` is the inverse of the function `xformByte` that you have defined in exercise 5.9.

We can now define the inverse S-box transform, using the multiplicative inverse function `gf28Inverse` you have defined in exercise 5.7:

```
InvSubByte : GF28 -> GF28
InvSubByte b = gf28Inverse (xformByte' b)

InvSubBytes : State -> State
InvSubBytes state = [ [ InvSubByte b | b <- row ]
                    | row <- state
                    ]
```

Exercise 5.21. Write and prove a Cryptol property showing that `InvSubByte` reverses `SubByte`.

Exercise 5.22. The AES specification provides an inverse S-box table [12, figure 14, section 5.3.2]. Write a Cryptol function `InvSubBytes'` using the table lookup technique. Make sure your implementation is correct (i.e., equivalent to `InvSubBytes`) by writing and proving a corresponding property.

5.9.2 The `InvShiftRows` transformation

The `InvShiftRows` transformation simply reverses the `ShiftRows` transformation from section 5.4:

```
InvShiftRows : State -> State
InvShiftRows state = [ row >>> shiftAmount
                     | row <- state
                     | shiftAmount <- [0 .. 3]
                     ]
```

Exercise 5.23. Write and prove a property stating that `InvShiftRows` is the inverse of `ShiftRows`.

5.9.3 The `InvMixColumns` transformation

Recall from section 5.5 that `MixColumns` amounts to matrix multiplication in GF(2^8). The inverse transform turns out to be the same, except with a different matrix:

```
InvMixColumns : State -> State
InvMixColumns state = gf28MatrixMult (m, state)
  where m = [[0x0e, 0x0b, 0x0d, 0x09],
             [0x09, 0x0e, 0x0b, 0x0d],
             [0x0d, 0x09, 0x0e, 0x0b],
             [0x0b, 0x0d, 0x09, 0x0e]]
```

Exercise 5.24. Write and prove a property stating that `InvMixColumns` is the inverse of `MixColumns`.

5.10 The inverse cipher

We now also have all the ingredients to encode AES decryption. Following figure 12 (section 5.3) of the AES standard [12]:

```
AESInvRound : (RoundKey, State) -> State
AESInvRound (rk, s) =
  InvMixColumns (AddRoundKey (rk, InvSubBytes (InvShiftRows s)))

AESFinalInvRound : (RoundKey, State) -> State
AESFinalInvRound (rk, s) = AddRoundKey (rk, InvSubBytes (InvShiftRows s))

aesDecrypt : ([128], [AESKeySize]) -> [128]
aesDecrypt (ct, key) = stateToMsg (AESFinalInvRound (kFinal, rounds ! 0))
  where
    (kFinal, ks, kInit) = ExpandKey key
    state0 = AddRoundKey(kInit, msgToState ct)
    rounds = [state0] # [ AESInvRound (rk, s)
                        | rk <- reverse ks
                        | s  <- rounds
                        ]
```

Note how we use the results of `ExpandedKey`, by carefully naming the first and last round keys and using the middle keys in reverse.

Testing Let us repeat the tests for AES encryption. Again, the first example comes from Appendix B of the AES standard [12]:

```
Cryptol> aesDecrypt (0x3925841d02dc09fbdc118597196a0b32,    \
                     0x2b7e151628aed2a6abf7158809cf4f3c)
0x3243f6a8885a308d313198a2e0370734
```

which agrees with the original value. The second example comes from Appendix C.1:

```
Cryptol> aesDecrypt (0x69c4e0d86a7b0430d8cdb78070b4c55a,   \
                     0x000102030405060708090a0b0c0d0e0f)
0x00112233445566778899aabbccddeeff
```

Again, the result agrees with the standard.

Other key sizes Similar to encryption, all we need to obtain AES192 decryption is to set `Nk` to be 6 in section 5.1. Setting `Nk` to 8 will correspondingly give us AES256.

The code You can see all the Cryptol code for AES in Appendix D.

5.11 Correctness

While test vectors do provide good evidence of AES working correctly, they do not provide a proof that we have implemented the standard faithfully. In fact, for a block cipher like AES, it is not possible to state what correctness would mean. Tweaking some parameters, or changing the S-box appropriately can give us a brand new cipher. And it would be impossible to tell this new cipher apart from AES aside from running it against published test vectors.

What we can do, however, is gain assurance that our implementation demonstrably has the desired properties. We have done this throughout this chapter by stating and proving a number of properties about AES and its constituent parts. The Cryptol methodology allows us to construct the code together with expected properties, allowing high-assurance development. We conclude this chapter with one final property, stating that `aesEncrypt` and `aesDecrypt` do indeed form an encryption-decryption pair:

```
property AESCorrect msg key = aesDecrypt (aesEncrypt (msg, key), key) == msg
```

Can we hope to automatically prove this theorem? For 128-bit AES, the state space for this theorem has 2^{256} elements. It would be naive to expect that we can prove this theorem by a push-button tool very quickly.[1] We can however, gain some assurance by running it through the `:check` command:

```
Using random testing.
Passed 100 tests.
Expected test coverage: 0.00% (100 of 2^^256 values)
```

You will notice that even running quick-check will take a while for the above theorem, and the total state space for this function means that we have not even scratched the surface! That said, being able to specify these properties together with very high level code is what distinguishes Cryptol from other languages when it comes to cryptographic algorithm programming.

[1] Note that, for a general algorithm with this large a state space, it is entirely possible to perform automatic verification using modern solvers, but if one carefully reflects upon the nature of cryptographic functions, it becomes clear why it should *not* be the case here.

Appendix A

Solutions to selected exercises

As with any language, there are usually multiple ways to write the same function in Cryptol. We have tried to use the most idiomatic Cryptol code segments in our solutions. Note that Cryptol prints numbers out in hexadecimal by default. In most of the answers below, we have implicitly used the command `:set base=10` to print numbers out in decimal for readability.

Section 1.2 Bits: Booleans (p. 1)

Exercise 1.1. (p. 1) Here is the response from Cryptol, in order:

```
True
[error] at <interactive>:1:1--1:6 Value not in scope: false
False
0x4
True
True
True
False
```

Section 1.3 Words: Numbers (p. 2)

Exercise 1.2. (p. 2) Cryptol prints the error "`Unsolvable constraint: 4 >= 5`", because `19` requires at least 5 bits to represent. `255:[8]` is the largest value of type `[8]`. `[6]` is the smallest type for `32`. `[0]` is the smallest type for `0`.

Exercise 1.3. (p. 2) `0xfeedfacef00d`.

Section 1.4 Integers: Unbounded numbers (p. 2)

Exercise 1.4. (p. 3) Here are Cryptol's responses:

```
0x1
17
11
```

`0x8 + 0x9` evaluates to `0x1` because addition on 4-bit words wraps around when the sum is 16 or more and it overflows. On the other hand, `Integer` addition never overflows, so the answer is 17. The final result is also due to wraparound: `- 0x5` evaluates to `0xb` or 11, which is $-5+16$.

Exercise 1.5. (p. 3) Here are Cryptol's responses:

```
4
11
4
```

Section 1.7 Tuples: Heterogeneous collections (p. 4)

Exercise 1.6. (p. 4) Here are Cryptol's responses:

```
(1, 6)
(True, False, True)
((1, 2), False, (2, (4, True)))
```

Exercise 1.7. (p. 4) Here are Cryptol's responses:

```
Cryptol> (1, 2+4).0
1
Cryptol> (1, 2+4).1
6
Cryptol> ((1, 2), False, (3-1, (4, True))).2.1
(4, True)
```

The required expression would be:

```
((1, 2), (2, 6, (True, 4)), False).1.2.0
```

Section 1.8 Sequences: Homogeneous collections (p. 4)

Exercise 1.9. (p. 4) In each case we get a type error:

```
Cryptol> [True, [True]]
[error] at <interactive>:1:8--1:14:
  Type mismatch:
    Expected type: Bit
    Inferred type: [1]
Cryptol> [[1, 2, 3], [4, 5]]
[error] at <interactive>:1:13--1:19:
  Type mismatch:
    Expected type: 3
    Inferred type: 2
```

In the first case, we are trying to put a bit (`True`) and a singleton sequence containing a bit (`[True]`) in the same sequence, which have different types. In the second case, we are trying to put two sequences of different lengths into a sequence, which again breaks the homogeneity requirement.

Exercise 1.10. (p. 5) Here are the responses from Cryptol:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 3, 5, 7, 9]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
[]
[10, 7, 4, 1]
[]
```

Note how `[10, 11 .. 1]` and `[10, 9 .. 20]` give us empty sequences, since the upper bound is smaller than the lower bound in the former, and larger in the latter.

Exercise 1.11. (p. 5) Here are the responses from Cryptol:

```
[(1, 4) (1, 5) (2, 4) (2, 5) (3, 4) (3, 5)]
[]
[(2, 3) (2, 4) (3, 3) (3, 4)]
```

The size of the result will be the sizes of the components multiplied. For instance, in the first example, the generator `x <- [1 .. 3]` assigns 3 values to `x`, and the generator `y <- [4, 5]` assigns 2 values to `y`; and hence the result has $2 \times 3 = 6$ elements.

Exercise 1.12. (p. 5) Here are the responses from Cryptol:

```
[(1, 4) (2, 5)]
[]
[(2, 3)]
```

In this case, the size of the result will be the minimum of the component sizes. For the first example, the generator `x <- [1 .. 3]` assigns 3 values to `x`, and the generator `y <- [4, 5]` assigns 2 values to `y`; and hence the result has $\min(2,3) = 2$ elements.

Exercise 1.13. (p. 6) Here is one way of writing such an expression, laid out in multiple lines to show the structure:

```
[ [ (i, j) | j <- [1 .. 3] ]
   | i <- [1 .. 3]
]
produces:
[[(1, 1), (1, 2), (1, 3)],
 [(2, 1), (2, 2), (2, 3)],
 [(3, 1), (3, 2), (3, 3)]]
```

The outer comprehension is a comprehension (and hence is nested). In particular the expression is:

```
[ (i, j) | j <- [1 .. 3] ]
```

You can enter the whole expression in Cryptol all in one line, or recall that you can put \ at line ends to continue to the next line. If you are writing such an expression in a program file, then you can lay it out as shown above or however most makes sense to you.

Exercise 1.14. (p. 6) Here are Cryptol's responses:

```
Cryptol> :set warnDefaulting=off
Cryptol> [] # [1, 2]
[1, 2]
Cryptol> [1, 2] # []
[1, 2]
Cryptol> [1 .. 5] # [3, 6, 8]
[1, 2, 3, 4, 5, 3, 6, 8]
Cryptol> [0 .. 9] @ 0
0
```

```
Cryptol> [0 .. 9] @ 5
5
Cryptol> [0 .. 9] @ 10
invalid sequence index: 10
Cryptol> [0 .. 9] @@ [3, 4]
[3, 4]
Cryptol> [0 .. 9] @@ []
[]
Cryptol> [0 .. 9] @@ [9, 12]
invalid sequence index: 12
Cryptol> [0 .. 9] @@ [9, 8 .. 0]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
Cryptol> [0 .. 9] ! 0
9
Cryptol> [0 .. 9] ! 3
6
Cryptol> [0 .. 9] !! [3, 6]
[6, 3]
Cryptol> [0 .. 9] !! [0 .. 9]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
Cryptol> [0 .. 9] ! 12
invalid sequence index: 12
```

Exercise 1.15. (p. 6) Using a permutation operator, we can simply write:

```
[0 .. 10] @@ [0, 2 .. 10]
```

Using a comprehension, we can express the same idea using:

```
[ [0 .. 10] @ i | i <- [0, 2 .. 10] ]
```

Strictly speaking, permutation operations are indeed redundant. However, they lead to more concise and easier-to-read expressions.

Exercise 1.16. (p. 6) When you type in an infinite sequence, Cryptol will only print the first 5 elements of it and will indicate that it is an infinite value by putting ... at the end.[2] Here are the responses:

```
[1, 2, 3, 4, 5, ...]
[1, 3, 5, 7, 9, ...]
2001
[601, 1001, 1401]
[100, 102, 104, 106, 108, ...]
```

Exercise 1.17. (p. 7) Here is a simple test case:

```
Cryptol> [1:[32] ...] ! 3

[error] at <interactive>:1:1--1:17:
  Expected a finite type, but found `inf`.
```

The error message is telling us that we *cannot* apply the reverse index operator (`!`) on an infinite sequence (`inf`). This is a natural consequence of the fact that one can never reach the end of an infinite sequence to count backwards. It is important to emphasize that this is a *type error*, i.e., the user gets this message at compile time; instead of Cryptol going into an infinite loop to reach the end of an infinite sequence.

Exercise 1.18. (p. 7) Here are Cryptol's responses:

[2]You can change this behavior by setting the `infLength` variable, like so: `Cryptol> :set infLength=10` will show the first 10 elements of infinite sequences


```
Cryptol> take`{3} [1 .. 12]
[1, 2, 3]
Cryptol> drop`{3} [1 .. 12]
[4, 5, 6, 7, 8, 9, 10, 11, 12]
Cryptol> split`{3}[1 .. 12]
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
Cryptol> groupBy`{3} [1 .. 12]
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
Cryptol> join [[1 .. 4], [5 .. 8], [9 .. 12]]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Cryptol> join [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Cryptol> transpose [[1, 2, 3, 4], [5, 6, 7, 8]]
[[1, 5], [2, 6], [3, 7], [4, 8]]
Cryptol> transpose [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

And for the fun bonus problems:

```
Cryptol> join [1,1]
Showing a specific instance of polymorphic result:
  * Using '1' for type argument 'each' of 'Cryptol::join'
3
Cryptol> transpose [1,2]
Showing a specific instance of polymorphic result:
  * Using '2' for type argument 'cols' of 'Cryptol::transpose'
[1, 2]
```

The argument to `join` must be a sequence of sequences, so each `1` must have a bitvector type. Cryptol defaults to the smallest possible type, `[1]`. So `join [1,1]` becomes `join [[True], [True]]`, which equals `[True, True]`, or `3`.

With `transpose`, Cryptol similarly chooses the smallest possible bitvector type `[2]` for the sequence elements. Then `[1,2]` becomes `[[False, True], [True, False]]`, which transposes to the same value.

Exercise 1.19. (p. 7) The following equalities are the simplest candidates:

```
join (split`{parts=n} xs) == xs
join (groupBy`{each=n} xs) == xs
split`{parts=n} xs == groupBy`{each=m} xs
transpose (transpose xs) == xs
```

In the first two equalities `n` must be a divisor of the length of the sequence `xs`. In the third equation, `n` $\times$ `m` must equal the length of the sequence `xs`.

Exercise 1.20. (p. 7) Append (`#`) joins two sequences of arbitrary length, while `join` appends a sequence of equal length sequences. In particular, the equality:

```
join [xs0, xs1, xs2, .. xsN] = xs0 # xs1 # xs2 ... # xsN
```

holds for all equal length sequences `xs0`, `xs1`, ..., `xsN`.

Exercise 1.21. (p. 8) Here they are:

```
Cryptol> split [1..12] : [4][3][8]
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
Cryptol> split [1..12] : [6][2][8]
[[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]]
Cryptol> split [1..12] : [12][1][8]
[[1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]]
```

Exercise 1.22. (p. 8) Cryptol will issue a type error:

```
Cryptol> split [1..12] : [5][2][8]
[error] at <interactive>:1:7--1:14:
  Type mismatch:
    Expected type: 10
    Inferred type: 12
```

Cryptol is telling us that we have requested 10 elements in the final result (5×2), but the input has 12.

Exercise 1.23. (p. 8) We can split 120 elements first into 3×40, splitting each of the the elements (`level1` below) into 4×10. A nested comprehension fits the bill:

```
[ split level1 : [4][10][8]
| level1 <- split ([1 .. 120] : [120][8]) : [3][40][8]
]
```

(Note again that you can enter the above in the command line all in one line, or by putting the line continuation character `\` at the end of the first two lines.)

Exercise 1.24. (p. 8) Here are Cryptol's responses:

```
[0, 0, 1, 2, 3]
[0, 0, 0, 0, 0]
[3, 4, 5, 0, 0]
[0, 0, 0, 0, 0]
[4, 5, 1, 2, 3]
[1, 2, 3, 4, 5]
[3, 4, 5, 1, 2]
[1, 2, 3, 4, 5]
```

Exercise 1.25. (p. 8) Rotating (left or right) by a multiple of the size of a sequence will leave it unchanged.

Section 1.9 Words as sequences (p. 8)

Exercise 1.26. (p. 8) Cryptol is big-endian, meaning that the most-significant-bit comes first. In the sequence `[True, False, True, False, True, False]`, the first element corresponds to the most-significant-digit, i.e., 2^5, the next element corresponds to the coefficient of 2^4, etc. A `False` bit yields a coefficient of 0 and a `True` bit gives 1. Hence, we have:

$$1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 32 + 0 + 8 + 0 + 2 + 0 = 42$$

Exercise 1.27. (p. 8) After issuing `:set base=2`, here are Cryptol's responses:

```
0b1100
0b11000
0b1100
0b101100
0b110001
0b100000
0b1100100000
True
```

Exercise 1.28. (p. 8) Remember that Cryptol is big-endian and hence `12:[6]` is precisely `[False, False, True, True, False, False]`. Here are Cryptol's responses:


```
Cryptol> take`{3} 0xFF
7
Cryptol> take`{3} (12:[6])
1
Cryptol> drop`{3} (12:[6])
4
Cryptol> split`{3} (12:[6])
[0, 3, 0]
Cryptol> groupBy`{3} (12:[6])
[1, 4]
```

For instance, the expression `take`{3} (12:[6])` evaluates as follows:

```
take`{3} (12:[6])
  = take (3, [False, False, True, True, False, False])
  = [False, False, True]
  = 1
```

Follow similar lines of reasoning to justify the results for the remaining expressions.

Exercise 1.29. (p. 9) Because of the leading zeros in `12:[12]`, they all produce different results:

```
Cryptol> take`{3} (12:[12])
0
Cryptol> drop`{3} (12:[12])
12
Cryptol> split`{3} (12:[12])
[0, 0, 12]
Cryptol> groupBy`{3} (12:[12])
[0, 0, 1, 4]
```

We will show the evaluation steps for `groupBy` here, and urge the reader to do the same for `split`:

```
groupBy`{3} (12:[12])
  = groupBy`{3} [False, False, False, False, False, False,
                 False, False, True, True, False, False]
  = [[False, False, False], [False, False, False]
     [False, False, True], [True, False, False]]
  = [0, 0, 1, 4]
```

Exercise 1.30. (p. 9) Here are Cryptol's responses:

```
3
48
```

Section 1.11 Records: Named collections (p. 10)

Exercise 1.31. (p. 10) Here are Cryptol's responses:

```
Cryptol> {xCoord = 12:[32], yCoord = 21:[32]}
{xCoord = 12, yCoord = 21}
Cryptol> {xCoord = 12:[32], yCoord = 21:[32]}.yCoord
21
Cryptol> {name = "Cryptol", address = "Galois"}
{name = "Cryptol", address = "Galois"}
```

```
Cryptol> {name = "Cryptol", address = "Galois"}.address
"Galois"
Cryptol> {name = "test", coords = {xCoord = 3:[32], yCoord = 5:[32]}}
{name = "test", coords = {xCoord = 3, yCoord = 5}}
Cryptol> {name = "test", coords = {xCoord = 3:[32], \
                                  yCoord = 5:[32]}}.coords.yCoord
5
Cryptol> {x=True, y=False} == {y=False, x=True}
True
```

Section 1.12 The `zero` (p. 10)

Exercise 1.32. (p. 11) Here are Cryptol's responses:

```
Cryptol> (zero : ([8] -> [3])) 5
0
Cryptol> (zero : Bit -> {xCoord : [12], yCoord : [5]}) True
{xCoord=0, yCoord=0}
```

The `zero` function returns `0`, ignoring its argument.

Section 1.13 Arithmetic (p. 11)

Exercise 1.33. (p. 11) Since `1` requires only 1 bit to represent, the result also has 1 bit. In other words, the arithmetic is done modulo $2^1 = 2$. Therefore, `1+1 = 0`.

Exercise 1.34. (p. 11) Now we have 8 bits to work with, so the result is `2`. Since we have 8 bits to work with, overflow will not happen until we get a sum that is at least 256.

Exercise 1.35. (p. 11) Recall from section 1.3 that there are no negative numbers in Cryptol. The values `3` and `5` can be represented in 3 bits, so Cryptol uses 3 bits to represent the result, so the arithmetic is done modulo $2^3 = 8$. Hence, the result is `6`. In the second expression, we have 8 bits to work with, so the modulus is $2^8 = 256$; so the subtraction results in `254` (or `0xfe`).

Exercise 1.36. (p. 11) The first, second, and fourth examples give a "division by 0" error message. In the third example, the division by zero occurs only in an unreachable branch, so the result is not an error.

In the last expression, the number `25` fits in 5 bits, so the modulus is $2^5 = 32$. The unary minus yields `7`, hence the result is `3`. Note that `lg2` is the *floor log base 2* function.

Exercise 1.37. (p. 11) Here are Cryptol's answers:

```
(2, 0)
(2, 1)
(2, 2)
(3, 0)
```

The following equation holds regarding `/` and `%`:

$$x = (x/y) * y + (x\%y)$$

whenever $y \neq 0$.

Exercise 1.38. (p. 12) The bit-width in this case is 3 (to accommodate for the number 5), and hence arithmetic is done modulo $2^3 = 8$. Thus, `-2` evaluates to `6`, leading to the result `min 5, (-2) == 5`. The parentheses are necessary because unary negation is handled in Cryptol's parser, not in its lexer, because whitespace is ignored. If this were not the case, reflect upon how you would differentiate the expressions `min 5 - 2` and `min 5 -2`.

Exercise 1.39. (p. 12) This time we are telling Cryptol to use precisely 8 bits, so `-2` is equivalent to `254`. Therefore the result is `254`.

Exercise 1.40. (p. 12) The idiomatic Cryptol way of summing two sequences is to use a comprehension:

```
[ i+j | i <- [1 .. 10]
      | j <- [10, 9 .. 1]
]
```

However, you will notice that the following will work as well:

```
[1 .. 10] + [10, 9 .. 1]
```

That is, Cryptol automatically lifts arithmetic operators to sequences, element-wise. However, it is often best to keep the explicit style of writing the comprehension, even though it is a bit longer, since that makes it absolutely clear what the intention is and how the new sequence is constructed, without depending implicitly upon Cryptol's automatic lifting.

Exercise 1.41. (p. 12) Here are Cryptol's responses:

```
Cryptol> ((2 >= 3) || (3 < 6)) && (4 == 5)
False
Cryptol> if 3 >= 2 then True else 1 < 12
True
```

Exercise 1.42. (p. 12) Here are Cryptol's responses:

```
[0, 1, 0, 1, 0, ...]
[0, 0, 0, 0, 0, ...]
```

The first example gets an element type of `[1]`, because of the explicit use of the numeral `1`, while the second example's element type is `[0]`. This is one case where `[k...]` is subtly different from `[k, k+1...]`.

Exercise 1.43. (p. 12) The expression `[1 .. 10]` is equivalent to `[1, (1+1) .. 10]`, and Cryptol knows that `10` requires at least 4 bits to represent and uses the minimum implied by all the available information. Hence we get: `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`. You can use the `:t` command to see the type Cryptol infers for this expression explicitly:

```
Cryptol> :t [1 .. 10]
{a} (a >= 4, fin a) => [10][a]
```

Cryptol tells us that the sequence has precisely 10 elements, and each element is at least 4 bits wide.

Section 1.14 Types (p. 13)

Exercise 1.44. (p. 13) We have 12 elements, each of which is a sequence of 3 elements; so we have $12 * 3 = 36$ elements total. Each element is a 6-bit word; so the total number of bits is $36 * 6 = 216$.

Exercise 1.45. (p. 13) `[inf][inf][32]`. The size of such a value would be infinite!

Exercise 1.46. (p. 15) Here is the type of `split`:

```
Cryptol> :t split
split : {parts, each, a} (fin each) =>
          [parts * each]a -> [parts][each]a
```

At every use case of `split` we must instantiate the parameters `parts`, `each`, and `a`, so that the resulting instantiation will match the use case. In the first example, we can simply take: `parts = 3`, `each = 3`, and `a = [4]`. In the second, we can take `parts=3`, `each=4`, and `a=[4]`. The third expression does not type-check. Cryptol tells us:

```
Cryptol> split`{3} [1..10] : [3][2][8]
[error] at <interactive>:1:11--1:18:
  Type mismatch:
    Expected type: 6
    Inferred type: 10
```

In this case, we are telling Cryptol that `parts = 3`, `each = 2`, and `a = [8]` by providing the explicit type signature. Using this information, Cryptol must ensure that the argument matches the instantiation of the the polymorphic type. The argument to `split` must have type `[parts * each]a`; instantiating the type variables, this is `[3 * 2][8]`, or `[6][8]`. However, the given argument has length `10`, which is not equal to `6`. It is not hard to see that there is no instantiation to make this work, since `10` is not divisible by `3`.

Exercise 1.47. (p. 16) Here is one way of writing this predicate, following the fact that $128 = 2 * 64$, $192 = 3 * 64$, and $256 = 4 * 64$:

```
{k} (2 <= k, k <= 4) => [k*64]
```

Here is another way, more direct but somewhat less satisfying:

```
{k} ((k - 128) * (k - 192) * (k - 256) == 0) => [k]
```

Note that Cryptol's type constraints do not include *or* predicates, hence we cannot just list the possibilities in a list.

Section 1.15 Defining functions (p. 16)

Exercise 1.48. (p. 16) Here are some example uses of `increment`:

```
Cryptol> increment 3
4
Cryptol> increment 255
0
Cryptol> increment 912
  [error] at <interactive>:1:1--1:14:
    Unsolvable constraint: 8 >= 10
```

Note how type inference rejects application when applied to an argument of the wrong size: 912 is too big to fit into 8 bits.

Exercise 1.49. (p. 17) Any of the following definitions will have the desired type:

```
Cryptol> let increment (x : [8]) = x + 1
Cryptol> let increment x = (x : [8]) + 1
Cryptol> let increment x = x + (1 : [8])
Cryptol> let increment x = x + 1 : [8]
Cryptol> :t increment
increment : [8] -> [8]
```


Exercise 1.50. (p. 17) The signature indicates that `twoPlusXY` is a function that takes two 8-bit words as a tuple, and returns an 8-bit word.

Exercise 1.51. (p. 17) Here is the type Cryptol infers:

```
Cryptol> :t twoPlusXY
twoPlusXY : {a} (a >= 2, fin a) => ([a],[a]) -> [a]
```

That is, our function will actually work over arbitrary (finite) sized words, as long as they are at least 2 bits wide. The 2-bit requirement comes from the constant 2, which requires at least 2 bits to represent.

Exercise 1.52. (p. 17) Here is one way of defining this function:

```
minMax4 : {a} (Cmp a) => [4]a -> (a, a)
minMax4 [a, b, c, d] = (e, f)
    where   e = min a (min b (min c d))
            f = max a (max b (max c d))
```

Note that ill-typed arguments will be caught at compile time! So, the second invocation with the 5 element sequence will fail to type-check. The `Cmp a` constraint arises from the types of `min` and `max` primitives:

```
min, max : {a} (Cmp a) => a -> a -> a
```

Exercise 1.53. (p. 18) Using `reverse` and `tail`, `butLast` is easy to define:

```
butLast : {n, t} (fin n) => [n+1]t -> [n]t
butLast xs = reverse (tail (reverse xs))
```

Here is another way to define `butLast`:

```
butLast' : {count, x} (fin count) => [count+1]x -> [count]x
butLast' xs = take`{count} xs
```

The type signature sets `count` to the desired width of the output, which is one shorter than the width of the input:

```
Cryptol> butLast []

[error] at <interactive>:1:1--1:11:
  Unsolved constraint:
    0 >= 1
      arising from
      matching types
      at <interactive>:1:1--1:11
```

At first the error message might be confusing. What Cryptol is telling us that it deduced `count+1` must be `1`, which makes `count` have value `0`. But the `count+1` we gave it was 0, which is not greater than or equal to 1.

Finally, note that `butLast` requires a finite sequence as input, for obvious reasons, and hence the `fin n` constraint.

Exercise 1.54. (p. 18)

```
all f xs = [ f x | x <- xs ] == ~zero
```

Note how we apply `f` to each element in the sequence and check that the result consists of all `True`s, by using a complemented `zero`. If we pass `any` an empty sequence, then we will always get `False`:

```
Cryptol> any eqTen [] where eqTen x = x == 10
False
```

This is intuitively the correct behavior as well. The predicate is satisfied by *none* of the elements in the sequence, but `any` requires at least one.

Exercise 1.55. (p. 18)

```
any f xs = [ f x | x <- xs ] != zero
```

This time all we need to make sure is that the result is not `zero`, i.e., at least one of elements yielded `True`. If we pass `all` an empty sequence, then we will always get `False`:

```
Cryptol> any eqTen [] where eqTen x = x == 10
False
```

Again, this is the correct response since there are no elements in an empty sequence that can satisfy the predicate.

Section 1.16 Recursion and recurrences (p. 18)

Exercise 1.56. (p. 19)

```
isOdd, isEven : {n} (fin n, n >= 1) => [n] -> Bit
isOdd  x = if x == 0 then False else isEven (x - 1)
isEven x = if x == 0 then True  else isOdd  (x - 1)
```

The extra predicate we need to add is `n >= 1`. This constraint comes from the subtraction with 1, which requires at least 1 bit to represent.

Exercise 1.57. (p. 19) A number is even if its least least significant bit is `False`, and odd otherwise. Hence, we can define these functions as:

```
isOdd', isEven' : {n} (fin n, n >= 1) => [n] -> Bit
isOdd'  x = x ! zero
isEven' x = ~(x ! zero)
```

Note the use of `zero` which permits Cryptol to choose the width of the `0` constant appropriately.

Exercise 1.58. (p. 19) The type of `maxSeq` is:

```
maxSeq : {a, b} (fin a, fin b) => [a][b] -> [b]
```

It takes a sequence of words and returns a word of the same size. The suggested expressions produce `0`, `10`, and `10`, respectively.

Exercise 1.59. (p. 19) We can simply drop the selection of the last element (`! 0`), and write `maxSeq'` as follows:

```
maxSeq' : {a, b} (fin a,fin b) => [a][b] -> [1 + a][b]
maxSeq' xs = ys
    where ys = [0] # [ max x  y | x <- xs
                                | y <- ys
                     ]
```

Exercise 1.60. (p. 20) Here is one answer. Note that in this solution the width of the answer is specified in terms of the width of the elements, so is likely to overflow. You can prevent the overflow by explicitly specifying the width of the output.

```
sumAll : {n, a} (fin n, fin a) => [n][a] -> [a]
sumAll xs = ys ! 0
  where ys = [0] # [ x+y | x <- xs
                         | y <- ys
                   ]
```

In this code, the sequence `ys` contains the partial running sums. This is precisely the same pattern we have seen in Example 1.59. The output for the example calls are:

```
CrashCourse> sumAll []
0
CrashCourse> sumAll [1]
1
CrashCourse> sumAll [1, 2]
3
CrashCourse> sumAll [1, 2, 3]
2
CrashCourse> sumAll [1, 2, 3, 4]
2
CrashCourse> sumAll [1, 2, 3, 4, 5]
7
CrashCourse> sumAll [1 .. 100]
58
```

If we do not explicitly tell Cryptol how wide the result is, then it will pick the width of the input elements, which will cause overflow and be subject to modular arithmetic as usual. Experiment with different signatures for `sumAll`, to avoid the overflow automatically, to get the answers:

```
0
1
3
6
10
15
5050
```

Exercise 1.61. (p. 20) Using a fold, it is easy to write `elem`:

```
elem (x, xs) = matches ! 0
    where matches = [False] # [ m || (x == e) | e <- xs
                                              | m <- matches
                              ]
```

Note how we or (`||`) the previous result `m` with the current match, accumulating the result as we walk over the sequence. Starting with `False` ensures that if none of the matches succeed we will end up returning `False`. We have:

```
Cryptol> elem (2, [1..10])
True
Cryptol> elem (0, [1..10])
False
Cryptol> elem (10, [])
False
```

Exercise 1.62. (p. 21)

```
fibs : [inf][32]
fibs = [0, 1] # [ x+y | x <- fibs
                      | y <- drop`{1} fibs
                ]
```

In this case we use the sequence `[0, 1]` as the seed, and both branches recursively refer to the defined value `fibs`. In the second branch, we `drop` the first element to skip over the first element of the sequence, effectively pairing the previous two elements at each step. The nth fibonacci number is obtained by the expression `fibs @ n`:

```
Cryptol> fibs @ 3
2
```

```
Cryptol> fibs @ 4
3
Cryptol> take`{10} fibs
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Note that `fibs` is an infinite stream of 32 bit numbers, so it will eventually be subject to wrap-around due to modular arithmetic.

Section 1.17 Stream equations (p. 21)

Exercise 1.63. (p. 21)

```
xs input = as where
   as = [0x89, 0xAB, 0xCD, 0xEF] # new
   new = [ a ^ b ^ c | a <- as
                     | b <- drop`{2} as
                     | c <- input ]
```

Section 1.18 Type synonyms (p. 21)

Exercise 1.64. (p. 22) A point is on the a^{th} axis if its non-a^{th} components are 0. Hence we have:

```
type Point3D a = {x : [a], y : [a], z : [a]}

onAnAxis : {a} (fin a) => Point3D a -> Bit
onAnAxis p = onX || onY || onZ
  where onX = (p.y == 0) && (p.z == 0)
        onY = (p.x == 0) && (p.z == 0)
        onZ = (p.x == 0) && (p.y == 0)
```

Exercise 1.65. (p. 23) This code:

```
cmpRing x y z = if x == y then z else z+z
```

yields the inferred type:

```
cmpRing : {a, b} (Eq a, Ring b) => a -> a -> b -> b
```

Section 2.1 Caesar's cipher (p. 27)

Exercise 2.1. (p. 28) Here is the alphabet and the corresponding shift-2 Caesar's alphabet:

```
Cryptol> ['A'..'Z']
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
Cryptol> ['A'..'Z'] <<< 2
"CDEFGHIJKLMNOPQRSTUVWXYZAB"
```


We use a left rotate to get the characters lined up correctly, as illustrated above.

Exercise 2.2. (p. 28) Here are Cryptol's responses:

```
Cryptol> caesar (0, "ATTACKATDAWN")
"ATTACKATDAWN"
Cryptol> caesar (3, "ATTACKATDAWN")
"DWWDFNDWGDZQ"
Cryptol> caesar (12, "ATTACKATDAWN")
"MFFMOWMFPMIZ"
Cryptol> caesar (52, "ATTACKATDAWN")
"ATTACKATDAWN"
```

If the shift is a multiple of 26 (as in 0 and 52 above), the letters will cycle back to their original values, so encryption will leave the message unchanged. Users of the Caesar's cipher should be careful about picking the shift amount!

Exercise 2.3. (p. 28) The code is almost identical, except we need to use a right rotate:

```
dCaesar : {n} ([8], String n) -> String n
dCaesar (s, msg) = [ shift x | x <- msg ]
      where map     = ['A' .. 'Z'] >>> s
            shift c = map @ (c - 'A')
```

We have:

```
Cryptol> caesar (12, "ATTACKATDAWN")
"MFFMOWMFPMIZ"
Cryptol> dCaesar (12, "MFFMOWMFPMIZ")
"ATTACKATDAWN"
```

Exercise 2.4. (p. 28) For the Caesar's cipher, the only good shifts are 1 through 25, since shifting by 0 would return the plaintext unchanged, and any shift amount `d` that is larger than 26 and over is essentially the same as shifting by `d % 26` due to wrap around. Therefore, all it takes to break the Caesar cipher is to try the sizes 1 through 25, and see if we have a valid message. We can automate this in Cryptol by returning all possible plaintexts using these shift amounts:

```
attackCaesar : {n} (String n) -> [25](String n)
attackCaesar msg = [ dCaesar(i, msg) | i <- [1 .. 25] ]
```

If we apply this function to `JHLZHYJPWOLYPZDLHR`, we get:

```
Cryptol> :set ascii=on
Cryptol> attackCaesar "JHLZHYJPWOLYPZDLHR",
["IGKYGXIOVNKXOYCKGQ", "HFJXFWHNUMJWNXBJFP", "GEIWEVGMTLIVMWAIEO"
 "FDHVDUFLSKHULVZHDN", "ECGUCTEKRJGTKUYGCM", "DBFTBSDJQIFSJTXFBL"
 "CAESARCIPHERISWEAK", "BZDRZQBHOGDQHRVDZJ", "AYCQYPAGNFCPGQUCYI"
 "ZXBPXOZFMEBOFPTBXH", "YWAOWNYELDANEOSAWG", "XVZNVMXDKCZMDNRZVF"
 "WUYMULWCJBYLCMQYUE", "VTXLTKVBIAXKBLPXTD", "USWKSJUAHZWJAKOWSC"
 "TRVJRITZGYVIZJNVRB", "SQUIQHSYFXUHYIMUQA", "RPTHPGRXEWTGXHLTPZ"
 "QOSGOFQWDVSFWGKSOY", "PNRFNEPVCUREVFJRNX", "OMQEMDOUBTQDUEIQMW"
 "NLPDLCNTASPCTDHPLV", "MKOCKBMSZROBSCGOKU", "LJNBJALRYQNARBFNJT"
 "KIMAIZKQXPMZQAEMIS"]
```

If you skim through the potential ciphertexts, you will see that the 7^{th} entry is probably the one we are looking for. Hence the key must be 7. Indeed, the message is `CAESARCIPHERISWEAK`.

Exercise 2.5. (p. 28) No. Using two shifts d_1 and d_2 is essentially the same as using just one shift with the amount $d_1 + d_2$. Our attack function would work just fine on this schema as well. In fact, we wouldn't even have to know how many rounds of encryption was applied. Multiple rounds is just as weak as a single round when it comes to breaking the Caesar's cipher.

Exercise 2.6. (p. 28) In this case we will fail to find a mapping:

```
Cryptol> caesar (3, "12")
... index of 240 is out of bounds
(valid range is 0 thru 25).
```

What happened here is that Cryptol computed the offset `'1' - 'A'` to obtain the 8-bit index 240 (remember, modular arithmetic!), but our alphabet only has 26 entries, causing the out-of-bounds error. We can simply remedy this problem by allowing our alphabet to contain all 8-bit numbers:

```
caesar' : {n} ([8], String n) -> String n
caesar' (s, msg) = [ shift x | x <- msg ]
  where map     = [0 .. 255] <<< s
        shift c = map @ c
```

Note that we no longer have to subtract `'A'`, since we are allowing a much wider range for our plaintext and ciphertext. (Another way to put this is that we are subtracting the value of the first element in the alphabet, which happens to be 0 in this case! Consequently, the number of "good" shifts increase from 25 to 255.) The change in `dCaesar'` is analogous:

```
dCaesar' : {n} ([8], String n) -> String n
dCaesar' (s, msg) = [ shift x | x <- msg ]
  where  map     = [0 .. 255] >>> s
         shift c = map @ c
```

Section 2.2 Vigenère cipher (p. 28)

Exercise 2.7. (p. 28) Here is one way to define `cycle`, using a recursive definition:

```
cycle xs = xss
      where xss = xs # xss
```

We have:

```
Cryptol> cycle [1 .. 3]
[1, 2, 3, 1, 2, ...]
```

If we do not have the `n >= 1` predicate, then we can pass `cycle` the empty sequence, which would cause an infinite loop emitting nothing. The predicate `n >= 1` makes sure the input is non-empty, guaranteeing that `cycle` can produce the infinite sequence.

Exercise 2.8. (p. 29)

```
vigenere (key, pt) = join [ caesar (k - 'A', [c])
                          | c <- pt
                          | k <- cycle key
                        ]
```

Note the shift is determined by the distance from the letter `'A'` for each character. Here is the cipher in action:

```
Cryptol> vigenere ("CRYPTOL", "ATTACKATDAWN")
"CKRPVYLVUYLG"
```

Exercise 2.9. (p. 29) Following the lead of the encryption, we can rely on `dCaesar`:

```
dVigenere : {n, m} (fin n, n >= 1) =>
            (String n, String m) -> String m
dVigenere (key, pt) = join [ dCaesar (k - 'A', [c])
                             | c <- pt
                             | k <- cycle key
                           ]
```

The secret code is:

```
Cryptol> dVigenere ("CRYPTOL", "XZETGSCGTYCMGEQGAGRDEQC")
"VIGENERECANTSTOPCRYPTOL"
```

Exercise 2.10. (p. 29) All it takes is to decrypt using using the plaintext as the key and message as the cipherkey. Here is this process in action. Recall from the previous exercise that encrypting `ATTACKATDAWN` by the key `CRYPTOL` yields `CKRPVYLVUYLG`. Now, if an attacker knows that `ATTACKATDAWN` and `CKRPVYLVUYLG` form a pair, he/she can find the key simply by:

```
Cryptol> dVigenere ("ATTACKATDAWN", "CKRPVYLVUYLG")
"CRYPTOLCRYPT"
```

Note that this process will not always tell us what the key is precisely. It will only be the key repeated for the given message size. For sufficiently large messages, or when the key does not repeat any characters, however, it would be really easy for an attacker to glean the actual key from this information.

This trick works since the act of using the plaintext as the key and the ciphertext as the message essentially reverses the shifting process, revealing the shift amounts for each pair of characters. The same attack would essentially work for the Caesar's cipher as well, where we would only need one character to crack it.

Section 2.3 The atbash (p. 29)

Exercise 2.11. (p. 29) Using the reverse index operator, coding atbash is trivial:

```
atbash : {n} String n -> String n
atbash pt = [ alph ! (c - 'A') | c <- pt ]
    where alph = ['A' .. 'Z']
```

We have:

```
Cryptol> atbash "ATTACKATDAWN"
"ZGGZXPZGWZDM"
```

Exercise 2.12. (p. 29) Notice that decryption for atbash is precisely the same as encryption, the process is entirely the same. So, we do not have to write any code at all, we can simply define:

```
dAtbash : {n} String n -> String n
dAtbash = atbash
```

We have:

```
Cryptol> dAtbash "ZGYZHSRHHVOUWVXIBKGRMT"
"ATBASHISSELFDECRYPTING"
```

Section 2.4 Substitution ciphers (p. 29)

Exercise 2.13. (p. 29)

```
subst (key, pt) = [ key @ (p - 'A') | p <- pt ]
```

We have:

```
Cryptol> subst(substKey, "SUBSTITUTIONSSAVETHEDAY")
"NLJNUXULUXAINNFSOUROWFC"
```

Exercise 2.14. (p. 29)

```
invSubst (key, c) = candidates ! 0
  where candidates = [0] # [ if c == k then a else p
                           | k <- key
                           | a <- ['A' .. 'Z']
                           | p <- candidates
                           ]
```

The comprehension defining `candidates` uses a fold (see page 20). The first branch (`k <- key`) walks through all the key elements, the second branch walks through the ordinary alphabet (`a <- ['A' .. 'Z']`), and the final branch walks through the candidate match so far. At the end of the fold, we simply return the final element of `candidates`. Note that we start with `0` as the first element, so that if no match is found we get a `0` back.

Exercise 2.15. (p. 30)

```
dSubst: {n} (String 26, String n) -> String n
dSubst (key, ct) = [ invSubst (key, c) | c <- ct ]
```

We have:

```
Cryptol> dSubst (substKey, "FUUFHKFUWFGI")
"ATTACKATDAWN"
```

Exercise 2.16. (p. 30) No, with this key we cannot decrypt properly:

```
Cryptol> subst ("AAAABBBBCCCCDDDDEEEEFFFFGG", "HELLOWORLD")
"BBCCDFDECA"
Cryptol> dSubst ("AAAABBBBCCCCDDDDEEEEFFFFGG", "BBCCDFDECA")
"HHLLPXPTLD"
```

This is because the given key maps multiple plaintext letters to the same ciphertext letter. (For instance, it maps all of `A`, `B`, `C`, and `D` to the letter `A`.) For substitution ciphers to work the key should not repeat the elements, providing a 1-to-1 mapping. This property clearly holds for `substKey`. Note that there is no shortage of keys, since for 26 letters we have 26! possible ways to choose keys, which gives us over 4-billion different choices.

Section 2.5 The scytale (p. 30)

Exercise 2.17. (p. 31) If you do not provide a signature for `msg'`, you will get the following type-error message from Cryptol:

```
Failed to validate user-specified signature.
  In the definition of 'scytale', at classic.cry:40:1--40:8:
    for any type row, diameter
      fin row
      fin diameter
    =>
    fin ?b
      arising from use of expression split at classic.cry:42:17--42:22
```

```
fin ?d
  arising from use of expression join at classic.cry:40:15--40:19
row * diameter == ?a * ?b
  arising from matching types at classic.cry:1:1--1:1
```

Essentially, Cryptol is complaining that it was asked to do a `split` and it figured that the constraint $diameter * row = a * b$ must hold, but that is not sufficient to determine what `a` and `b` should really be. (There could be multiple ways to assign `a` and `b` to satisfy that requirement, for instance `a=4`, `b=row`; or `a=2` and `b=2*row`, resulting in differing behavior.) This is why it is unable to "validate the user-specified signature". By putting the explicit signature for `msg'`, we are giving Cryptol more information to resolve the ambiguity. Notice that since the code for `scytale` and `dScytale` are precisely the same except for the type on `msg'`. This is a clear indication that the type signature plays an essential role here.

Exercise 2.18. (p. 31) Even if we do not know the diameter, we do know that it is a divisor of the length of the message. For any given message size, we can compute the number of divisors of the size and try decryption until we find a meaningful plaintext. Of course, the number of potential divisors will be large for large messages, but the practicality of scytale stems from the choice of relatively small diameters, hence the search would not take too long. (With large diameters, the ancient Greeks would have to carry around very thick rods, which would not be very practical in a battle scenario!)

Section 3.1 The plugboard (p. 33)

Exercise 3.1. (p. 33) We can simply ask Cryptol what the implied mappings are:

```
Cryptol> [ plugboard @ (c - 'A') | c <- "ACQTUWO" ]
"HGXVYML"
```

Why do we subtract the `'A'` when indexing?

Section 3.2 Scrambler rotors (p. 33)

Exercise 3.2. (p. 34) Recall that `rotor1` was defined as:

```
rotor1 = mkRotor ("RJICAWVQZODLUPYFEHXSMTKNGB", "IO")
```

Here is a listing of the new mappings and the characters we will get at the output for each successive `C`:

starting map	output	notch engaged?
RJICAWVQZODLUPYFEHXSMTKNGB	I	no
JICAWVQZODLUPYFEHXSMTKNGBR	C	no
ICAWVQZODLUPYFEHXSMTKNGBRJ	A	yes
CAWVQZODLUPYFEHXSMTKNGBRJI	W	no
AWVQZODLUPYFEHXSMTKNGBRJIC	V	no

Note how we get different letters as output, even though we are providing the same input (all `C`'s.) This is the essence of the Enigma: the same input will not cause the same output necessarily, making it a polyalphabetic substitution cipher.

Section 3.3 Connecting the rotors: notches in action (p. 34)

Exercise 3.3. (p. 35) We can define the following value to simulate the operation of always telling `scramble` to rotate the rotor and providing it with the input `C`.

```
rotor1CCCCC = [(c1, n1), (c2, n2), (c3, n3), (c4, n4), (c5, n5)]
   where (n1, c1, r1) = scramble (True, 'C', rotor1)
         (n2, c2, r2) = scramble (True, 'C', r1)
         (n3, c3, r3) = scramble (True, 'C', r2)
         (n4, c4, r4) = scramble (True, 'C', r3)
         (n5, c5, r5) = scramble (True, 'C', r4)
```

Note how we chained the output rotor values in the calls, through the values `r1-r2-r3` and `r4`. We have:

```
Cryptol> rotor1CCCCC
[(I, False), (C, False), (A, True), (W, False), (V, False)]
```

Note that we get precisely the same results from Cryptol as we predicted in the previous exercise.

Exercise 3.4. (p. 36) Not unless we receive an empty sequence of rotors, i.e., a call of the form: `joinRotors ([], c)` for some character `c`. In this case, it does make sense to return `c` directly, which is what `initRotor` will do. Note that unless we do receive an empty sequence of rotors, the value of `initRotor` will not be used when computing the `joinRotors` function.

Exercise 3.5. (p. 36) The crucial part is the value of `ncrs`. Let us write it out by substituting the values of `rotors` and `inputChar`:

```
ncrs = [(True, 'F', initRotor)]
         # [  scramble (notch, char, r)
            | r <- [rotor1, rotor2, rotor3]
            | (notch, char, rotor') <- ncrs
            ]
```

Clearly, the first element of `ncrs` will be:

```
(True, 'F', initRotor)
```

Therefore, the second element will be the result of the call:

```
scramble (True, 'F', rotor1)
```

Recall that `rotor1` was defined as:

```
rotor1 = mkRotor ("RJICAWVQZODLUPYFEHXSMTKNGB", "IO")
```

What letter does `rotor1` map `F` to? Since `F` is the 5th character (counting from 0), `rotor1` maps it to the 5th element of its permutation, i.e., `W`, remembering to count from 0! The topmost element in `rotor1` is `R`, which is not its notch-list, hence it will *not* tell the next rotor to rotate. But it will rotate itself, since it received the `True` signal. Thus, the second element of `ncrs` will be:

```
(False, 'W', ...)
```

where we used `...` to denote the one left-rotation of `rotor1`. (Note that we do not need to know the precise arrangement of `rotor1` now for the purposes of this exercise.) Now we move to `rotor2`, we have to compute the result of the call:

```
scramble (False, 'W', rotor2)
```

Recall that `rotor2` was defined as:

```
rotor2 = mkRotor ("DWYOLETKNVQPHURZJMSFIGXCBA", "B")
```

So, it maps `W` to `X`. (The fourth letter from the end.) It will not rotate itself, and it will not tell `rotor3` to rotate itself either since the topmost element is `D` in its current configuration, and `D` which is not in the notch-list `"B"`. Thus, the final `scramble` call will be:

```
scramble (False, 'X', rotor3)
```

where

```
rotor3 = mkRotor ("FGKMAJWUOVNRYIZETDPSHBLCQX", "CK")
```

It is easy to see that `rotor3` will map `X` to `C`. Thus the final value coming out of this expression must be `C`. Indeed, we have:

```
Cryptol> project(2, 2, joinRotors ([rotor1 rotor2 rotor3], 'F'))
C
```

Of course, Cryptol also keeps track of the new rotor positions as well, which we have glossed over in this discussion.

Section 3.4 The reflector (p. 36)

Exercise 3.6. (p. 36)

```
all : {n, a} (fin n) => (a -> Bit) -> [n]a -> Bit
all f xs = [ f x | x <- xs ] == ~zero

checkReflector refl = all check ['A' .. 'Z']
  where check c = (c != m) && (c == c')
          where   m  = refl @ (c - 'A')
                  c' = refl @ (m - 'A')
```

For each character in the alphabet, we first figure out what it maps to using the reflector, named `m` above. We also find out what `m` gets mapped to, named `c'` above. To be a valid reflector it must hold that `c` is not `m` (no character maps to itself), and `c` must be `c'`. We have:

```
Cryptol> checkReflector reflector
True
```

Note how we used `all` to make sure `check` holds for all the elements of the alphabet.

Section 3.5 Putting the pieces together (p. 36)

Exercise 3.7. (p. 37) We can define the following helper function, using the function `all` you have defined in exercise 1.54:

```
checkPermutation : Permutation -> Bit
checkPermutation perm = all check  ['A' .. 'Z']
  where check  c = (c == substBwd(perm, substFwd (perm, c)))
                && (c == substFwd(perm, substBwd (perm, c)))
```

Note that we have to check both ways (first `substFwd` then `substBwd`, and also the other way around) in case the substitution is badly formed, for instance if it is mapping the same character twice. We have:

```
Cryptol> checkPermutation [ c | (c, _) <- rotor1 ]
True
```

For a bad permutation we would get `False`:

```
Cryptol> checkPermutation (['A' .. 'Y'] # ['A'])
False
```

Exercise 3.8. (p. 37) Since the reflector is symmetric, substituting backwards or forwards does not matter. We can verify this with the following helper function:

```
all : {a, b} (fin b) => (a -> Bit) -> [b]a -> Bit
all fn xs = folds ! 0 where
  folds = [True] # [ fn x && p | x <- xs
                               | p <- folds]
checkReflectorFwdBwd : Reflector -> Bit
checkReflectorFwdBwd refl = all check ['A' .. 'Z']
  where check c = substFwd (refl, c) == substBwd (refl, c)
```

We have:

```
Cryptol> checkReflectorFwdBwd reflector
True
```

Section 3.7 Encryption and decryption (p. 38)

Exercise 3.10. (p. 39) Enigma will start repeating once the rotors go back to their original position. With n rotors, this will take 26^n characters. In the case of the traditional 3-rotor Enigma this amounts to $26^3 = 17576$ characters. Note that we are assuming an ideal Enigma here with no double-stepping [2].

Exercise 3.11. (p. 39) Since the period for the 3-rotor Enigma is 17576 (see the previous exercise), we need to make sure two instances of `CRYPTOL` are 17576 characters apart. Since `CRYPTOL` has 7 characters, we should have 17569 X's. The following Cryptol definition would return the relevant pieces:

```
enigmaCryptol = (take`{7} ct, drop`{17576} ct)
  where   str = "CRYPTOL" # [ 'X' | _ <- [1 .. 17569] ]
                # "CRYPTOL"
          ct  = dEnigma(modelEnigma, str)
```

We have:

```
Cryptol> enigmaCryptol
("KGSHMPK", "KGSHMPK")
```

As predicted, both instances of `CRYPTOL` get encrypted as `KGSHMPK`.

Section 4.1 Writing properties (p. 41)

Exercise 4.1. (p. 42)

```
property revRev xs = reverse (reverse xs) == xs
```

Exercise 4.2. (p. 42)

```
property appAssoc (xs, ys, zs) = xs # (ys # zs) == (xs # ys) # zs
```

Exercise 4.3. (p. 42)

```
property revApp (xs, ys) = reverse (xs # ys)
                           == reverse ys # reverse xs
```


Exercise 4.4. (p. 42)

```
property lshMul (n, k) = n << k == n * 2^^k
```

Exercise 4.5. (p. 43) There are many such types, all sharing the property that they do not take any space to represent. Here are a couple examples:

```
Cryptol> flipNeverIdentity (zero : ([0], [0]))
False
Cryptol> flipNeverIdentity (zero : [0][8])
False
```

Exercise 4.6. (p. 43)

```
Cryptol> :t widthPoly
widthPoly : {a, b} (fin a) => [a]b -> Bit
```

It is easy to see that `widthPoly` holds at the instances:

```
{b} [15]b -> Bit
```

and

```
{b} [531]b -> Bit
```

but at no other. Based on this, we can write `evenWidth` as follows:

```
property evenWidth x = (length x) ! 0 == False
```

remembering that the 0'th bit of an even number is always `False`. We have:

```
Cryptol> evenWidth (0:[1])
False
Cryptol> evenWidth (0:[2])
True
Cryptol> evenWidth (0:[3])
False
Cryptol> evenWidth (0:[4])
True
Cryptol> evenWidth (0:[5])
False
```

Section 4.2 Establishing correctness (p. 43)

Exercise 4.7. (p. 45) If we try to prove `revRev` directly, we will get an error from Cryptol:

```
Cryptol> :prove revRev
Not a monomorphic type:
{n, a} (Cmp a, fin n) => [n]a -> Bit
```

Cryptol is telling us that the property has a polymorphic type, and hence cannot be proven. We can easily prove instances of it, by either creating new properties with fixed type signatures, or by specializing it via type annotations. Several examples are given below:

```
Cryptol> :prove revRev : [10][8] -> Bit
Q.E.D.
Cryptol> :prove revRev : [100][32] -> Bit
Q.E.D.
Cryptol> :prove revRev : [0][4] -> Bit
Q.E.D.
```

Exercise 4.11. (p. 45) We have:

```
Cryptol> :prove widthPoly : [15] -> Bit
Q.E.D.
Cryptol> :prove widthPoly : [531] -> Bit
Q.E.D.
Cryptol> :prove widthPoly : [8] -> Bit
widthPoly:[8] -> Bit 0 = False
Cryptol> :prove widthPoly : [32] -> Bit
widthPoly:[32] -> Bit 0 = False
```

Exercise 4.12. (p. 45)

```
property divModMul (x,y) = if y == 0
                           then True   // precondition fails => True
                           else x == (x / y) * y + x % y
```

We have:

```
Cryptol> :prove divModMul : ([4], [4]) -> Bit
Q.E.D.
Cryptol> :prove divModMul : ([8], [8]) -> Bit
Q.E.D.
```

Exercise 4.13. (p. 46) Using `all` and `elem`, it is easy to express `validMessage`:

```
validMessage = all (\c -> elem (c, ['A' .. 'Z']))
```

Note the use of a λ-expression to pass the function to `all`. Of course, we could have defined a separate function for it in a `where`-clause, but the function is short enough to directly write it inline.

Exercise 4.14. (p. 46) A naive attempt would be to write:

```
property caesarCorrectBOGUS (d,msg) =
         dCaesar(d, caesar(d, msg)) == msg
```

However, this property is not correct for all `msg`, since Caesar's cipher only works for messages containing the letters `'A'` …`'Z'`, not arbitrary 8-bit values as the above property suggests. We can see this easily by providing a bad input:

```
Cryptol> caesar (3, "1")
invalid sequence index: 240
```

(240 is the difference between the ASCII value of `'1'`, 49, and the letter `'A'`, 65, interpreted as an 8-bit offset.) We should use the `validMessage` function of the previous exercise to write a conditional property instead:

```
property caesarCorrect (d,msg) = if validMessage msg
                                 then dCaesar(d, caesar(d, msg)) == msg
                                 else True
```

We have:

```
Cryptol> :prove caesarCorrect : ([8], String(10)) -> Bit
Q.E.D.
```

Exercise 4.15. (p. 46)

```
property modelEnigmaCorrect pt =
     if validMessage pt
     then dEnigma (modelEnigma, enigma (modelEnigma, pt)) == pt
     else True
```

We have:

```
Cryptol> :prove modelEnigmaCorrect : String(10) -> Bit
Q.E.D.
```

Section 4.3 Automated random testing (p. 46)

Exercise 4.16. (p. 46) Here is the interaction with Cryptol (when you actually run this, you will see the test cases counting up as they are performed):

```
Cryptol> :check (caesarCorrect : ([8], String 10) -> Bit)
Using random testing.
Passed 100 tests.
Expected test coverage: 0.00% (100 of 2^^88 values)
Cryptol> :set tests=1000
Cryptol> :check (caesarCorrect : ([8], String 10) -> Bit)
Using random testing.
Passed 1000 tests.
Expected test coverage: 0.00% (1000 of 2^^88 values)
```

In each case, Cryptol tells us that it checked a minuscule portion of all possible test cases: A good reminder of what `:check` is really doing. The number of test cases is: $2^{8+8\times10} = 2^{88}$. We have 8-bits for the `d` value, and $10 * 8$ bits total for the 10 characters in `msg`, giving us a total of 88 bits. Since the input is 88 bits wide, we have 2^{88} potential test cases. Note how the number of test cases increase exponentially with the size of the message.

Exercise 4.17. (p. 46)

```
Cryptol> :check True
Using exhaustive testing.
Passed 1 tests.
Q.E.D.
Cryptol> :check False
Using exhaustive testing.
False = False0%
Cryptol> :check \x -> x == (x:[8])
Using exhaustive testing.
Passed 256 tests.
Q.E.D.
```

Note that when Cryptol is able to exhaust all possible inputs, it returns `Q.E.D.`, since the property is effectively proven.

Exercise 4.18. (p. 46)

```
property easyBug x = x != (76123:[64])
```

The `:prove` command will find the counterexample almost instantaneously, while `:check` will have a hard time!

Exercise 4.19. (p. 47) The predicate passes 100 tests, with a test coverage of 0.15%.

```
Cryptol> :exhaust \(x:[16]) -> (x + 6) / 3 == x / 3 + 2
Using exhaustive testing.
(\(x : [16]) -> (x + 6) / 3 == x / 3 + 2) 0xfffa = False
```

The property does not hold in general.

Section 4.4 Checking satisfiability (p. 47)

Exercise 4.20. (p. 48)

```
modFermat (a, b, c, n) = (a > 0) && (b > 0) && (c > 0) && (n > 2)
                        && (a^^n + b^^n == c^^n)
```

The `fin s` predicate comes from the fact that we are doing arithmetic and comparisons. The predicate `s >= 2` comes from the fact that we are comparing `n` to 2, which needs at least 2 bits to represent.

Exercise 4.21. (p. 48) We can try different instantiations as follows:

```
Cryptol> :sat modFermat : Quad(2) -> Bit
modFermat : Quad(2) -> Bit (1, 2, 1, 3) = True
Cryptol> :sat modFermat : Quad(3) -> Bit
modFermat : Quad(3) -> Bit (4, 4, 4, 4) = True
Cryptol> :sat modFermat : Quad(4) -> Bit
modFermat : Quad(4) -> Bit (4, 4, 4, 8) = True
```

The modular form of Fermat's last theorem does not hold for any of the instances up to and including 12 bits wide, when I stopped experimenting myself. It is unlikely that it will hold for any particular bit-size, although the above demonstration is not a proof. (We would need to consult a mathematician for the general result!) Also note that Cryptol takes longer and longer to find a satisfying instance as you increase the bit-size.

Section 5.2 Polynomials in GF(2^8) (p. 50)

Exercise 5.1. (p. 50)

```
polySelfAdd: GF28 -> Bit
property polySelfAdd x = (x ^ x) == zero
```

We have:

```
Cryptol> :prove polySelfAdd
Q.E.D.
```

Exercise 5.2. (p. 51)

```
gf28Add ps = sums ! 0
  where sums = [zero] # [ p ^ s | p <- ps
                                | s <- sums
                        ]
```

Exercise 5.3. (p. 51) We first compute the results of multiplying our first polynomial $(x^3 + x^2 + x + 1)$ with each term in the second polynomial $(x^2 + x + 1)$ separately:

$$
\begin{array}{lcl}
(x^3 + x^2 + x + 1) \times x^2 & = & x^5 + x^4 + x^3 + x^2 \\
(x^3 + x^2 + x + 1) \times x & = & x^4 + x^3 + x^2 + x \\
(x^3 + x^2 + x + 1) \times 1 & = & x^3 + x^2 + x \ \ + 1
\end{array}
$$

We now add the resulting polynomials, remembering the adding the same powered terms cancel each other out. For instance, we have two instances each of x^4, x^2, and x, which all get canceled. We have three instances each of x^3 and x^2, so they survive the addition, etc. After the addition, we are left with the polynomial $x^5 + x^3 + x^2 + 1$, which can be interpreted as `0b00101101`, i.e., 45.

Exercise 5.4. (p. 51) The long division algorithm is laborious, but not particularly hard:

$$
\begin{array}{r|cccccc}
 & & & & x^2 & +x & \\
\hline
x^3 + x^2 + 1 & x^5 & & +x^3 & & & +1 \\
 & x^5 & +x^4 & & +x^2 & & \\
\hline
 & & x^4 & +x^3 & +x^2 & & +1 \\
 & & x^4 & +x^3 & & +x & \\
\hline
 & & & & x^2 & +x & +1
\end{array}
$$

Therefore, the quotient is $x^2 + x$ and the remainder is $x^2 + x + 1$. We can verify this easily with Cryptol:

```
Cryptol> pdiv <| x^^5 + x^^3 + 1 |> <| x^^3 + x^^2 + 1 |> == \
   <| x^^2 + x |>
True
Cryptol> pmod <| x^^5 + x^^3 + 1 |> <| x^^3 + x^^2 + 1 |> == \
   <| x^^2 + x + 1 |>
True
```

Another way to check your result would be to multiply the quotient by the divisor and add the remainder, and check that it gives us precisely the polynomial we started with:

```
Cryptol> pmult <| x^^2 + x |>  <| x^^3 + x^^2 + 1 |> \
         ^ <| x^^2 + x + 1 |>
0x29
Cryptol> <| x^^5 + x^^3 + 1 |>
0x29
```

Exercise 5.5. (p. 51)

```
property gf28MultUnit         x =      gf28Mult(x, 1) == x
property gf28MultCommutative  x y =    gf28Mult(x, y) == gf28Mult(y, x)
property gf28MultAssociative  x y z =  gf28Mult(x, gf28Mult(y, z))
                                       == gf28Mult(gf28Mult(x, y), z)
```

It turns out that proving the unit and commutativity are fairly trivial:

```
Cryptol> :prove gf28MultUnit
Q.E.D.
Cryptol> :prove gf28MultCommutative
Q.E.D.
```

But `aesMultAssociative` takes much longer! We show the results of `:check` below:

```
Cryptol> :check gf28MultAssociative
Checking case 1000 of 1000 (100.00%)
1000 tests passed OK
```

Note that the coverage is pretty small (on the order of 0.006%) in this case. Proving associativity of multiplication algorithms using SAT/SMT based technologies is a notoriously hard task [9, section 6.3.1]. If you have the time, you can let Cryptol run long enough to complete the `:prove gf28MultAssociative` command, however.

Section 5.3 The `SubBytes` transformation (p. 51)

Exercise 5.6. (p. 52)

```
gf28Pow (n, k) = pow k
   where   sq x  = gf28Mult (x, x)
           odd x = x ! 0
           pow i = if i == 0 then 1
                   else if odd i
                        then gf28Mult (n, sq (pow (i >> 1)))
                        else sq (pow (i >> 1))
```

Here is a version that follows the stream-recursion pattern:

```
gf28Pow' : (GF28, [8]) -> GF28
gf28Pow' (n, k) = pows ! 0
  where   pows = [1] # [ if bit then gf28Mult (n, sq x)
                                else sq x
                       | x <- pows
                       | bit <- k
                       ]
          sq x = gf28Mult (x, x)
```

Exercise 5.7. (p. 52)

```
gf28Inverse x = gf28Pow (x, 254)
```

We do not have to do anything special about `0`, since our `gf28Inverse` function yields `0` in that case:

```
Cryptol> gf28Inverse 0
0x00
```

Exercise 5.8. (p. 52) Since `0` does not have a multiplicative inverse, we have to write a conditional property:

```
property gf28InverseCorrect x =
    if x == 0 then x' == 0 else gf28Mult(x, x') == 1
  where x' = gf28Inverse x
```

We have:

```
Cryptol> :prove gf28InverseCorrect
Q.E.D.
```

Exercise 5.9. (p. 52)

```
xformByte b = gf28Add [b, (b >>> 4), (b >>> 5),
                       (b >>> 6), (b >>> 7), c]
  where c = 0x63
```

Exercise 5.10. (p. 53)


```
property SubByteCorrect x = SubByte x == SubByte' x
```

We have:

```
Cryptol> :prove SubByteCorrect
Q.E.D.
```

Section 5.4 The `ShiftRows` transformation (p. 53)

Exercise 5.11. (p. 53) Consider what happens after 4 calls to `ShiftRows`. The first row will stay the same, since it never moves. The second row moves one position each time, and hence it will move 4 positions at the end, restoring it back to its original starting configuration. Similarly, row 3 will rotate $2 \times 4 = 8$ times, again restoring it. Finally, row 4 rotates 3 times each for a total of $3 \times 4 = 12$ times, cycling back to its starting position. Hence, every 4th rotation will restore the entire state back. We can verify this in Cryptol by the following property:

```
shiftRow4RestoresBack : State -> Bit
property shiftRow4RestoresBack state = states @ 4 == state
   where states = [state] # [ ShiftRows s | s <- states ]
```

We have:

```
Cryptol> :prove shiftRow4RestoresBack
Q.E.D.
```

Of course, any multiple of 4 would have the same effect.

Section 5.5 The `MixColumns` transformation (p. 54)

Exercise 5.12. (p. 54)

```
gf28DotProduct (xs, ys) = gf28Add [ gf28Mult (x, y) | x <- xs
                                                    | y <- ys
                                  ]
```

Exercise 5.13. (p. 54)

```
property DPComm a b =     gf28DotProduct (a, b) == gf28DotProduct (b, a)
property DPDist a b c =   gf28DotProduct (a, vectorAdd(b, c)) ==
                          gf28Add [ab, ac]
   where   ab = gf28DotProduct (a, b)
           ac = gf28DotProduct (a, c)
           vectorAdd (xs, ys) = [ gf28Add [x, y] | x <- xs
                                                 | y <- ys
                                ]
```

We have:

```
AES> :prove DPComm : [10]GF28 -> [10]GF28 -> Bit
Q.E.D.
AES> :check DPDist : [10]GF28 -> [10]GF28 -> [10]GF28 -> Bit
Using random testing.
Passed 1000 tests.
```

You might be surprised that the total number of cases for this property is $2^{3*10*8} = 2^{240}$—a truly ginormous number!

Exercise 5.14. (p. 54)

```
gf28VectorMult (v, ms) = [ gf28DotProduct(v, m) | m <- ms ]
```

Exercise 5.15. (p. 54) We simply need to call `gfVectorMult` of the previous exercise on every row of the first matrix, after transposing the second matrix to make sure columns are properly aligned. We have:

```
gf28MatrixMult (xss, yss) = [ gf28VectorMult(xs, yss')
                            | xs <- xss ]
      where yss' = transpose yss
```

Section 5.6 Key expansion (p. 55)

Exercise 5.16. (p. 55) Finding out the elements is easy:

```
Cryptol> [ (Rcon i)@0 | i <- [1 .. 10] ]
[1, 2, 4, 8, 16, 32, 64, 128, 27, 54]
```

Note that we only capture the first element in each `Rcon` value, since we know that the rest are 0. We can now use this table to define `Rcon'` as follows:

```
Rcon' i = [(rcons @ (i-1)), 0, 0, 0]
     where rcons : [10]GF28
           rcons = [1, 2, 4, 8, 16, 32, 64, 128, 27, 54]
```

Note that we subtract 1 before indexing into the `rcons` sequence to get the indexing right.

Exercise 5.17. (p. 55) We need to write a conditional property (subsection 4.2.4). Below, we use the function `elem` you have defined in exercise 1.61:

```
property RconCorrect x = if elem(x, [1..10])
                         then Rcon x == Rcon' x
                         else True
```

We have:

```
Cryptol> :prove RconCorrect
Q.E.D.
```

Section 5.8 AES encryption (p. 57)

Exercise 5.18. (p. 58)

```
property msgToStateToMsg msg = stateToMsg(msgToState(msg)) == msg
property stToMsgToSt s = msgToState(stateToMsg s) == s
```

We have:

```
Cryptol> :prove msgToStateToMsg
Q.E.D.
Cryptol> :prove stToMsgToSt
Q.E.D.
```


Section 5.9 Decryption (p. 58)

Exercise 5.20. (p. 59)

```
property xformByteInverse x = xformByte' (xformByte x) == x
```

We have:

```
Cryptol> :prove xformByteInverse
Q.E.D.
```

Exercise 5.21. (p. 59)

```
property sboxInverse s = InvSubBytes (SubBytes s) == s
```

We have:

```
Cryptol> :prove xformByteInverse
Q.E.D.
```

Exercise 5.23. (p. 59)

```
property shiftRowsInverse s = InvShiftRows (ShiftRows s) == s
```

We have:

```
Cryptol> :prove shiftRowsInverse
Q.E.D.
```

Exercise 5.24. (p. 59)

```
property mixColumnsInverse s = InvMixColumns (MixColumns s) == s
```

Unlike others, this property is harder to prove automatically and will take much longer. Below we show the `:check` results instead:

```
Cryptol> :check mixColumnsInverse
Checking case 1000 of 1000 (100.00%)
1000 tests passed OK
```

Appendix B

Cryptol prelude functions

Bitwise and logical operations

```
True, False : Bit
&&, ||, ^   : {a} (Logic a) => a -> a -> a
~           : {a} (Logic a) => a -> a
==>, /\, \/ : Bit -> Bit -> Bit
```

Comparisons

```
==, !=            : {a} (Eq a) => a -> a -> Bit
<, >, <=, >=      : {a} (Cmp a) => a -> a -> Bit
<$, >$, <=$, >=$ : {a} (SignedCmp a) => a -> a -> Bit
min, max          : {a} (Cmp a) => a -> a -> a
===, !==          : {a, b} (Eq b) => (a -> b) -> (a -> b) -> a -> Bit
```

Arithmetic

```
+, -, *            : {a} (Ring a) => a -> a -> a
negate             : {a} (Ring a) => a -> a
fromInteger        : {a} (Ring a) => Integer -> a
^^                 : {a, e} (Ring a, Integral e) => a -> e -> a
abs                : {a} (Cmp a, Ring a) => a -> a
/, %               : {a} (Integral a) => a -> a -> a
toInteger          : {a} (Integral a) => a -> Integer
lg2                : {n} (fin n) => [n] -> [n]
/$, %$             : {n} (fin n, n >= 1) => [n] -> [n] -> [n]
carry              : {n} (fin n) => [n] -> [n] -> Bit
scarry, sborrow    : {n} (fin n, n >= 1) => [n] -> [n] -> Bit
zext               : {m, n} (fin m, m >= n) => [n] -> [m]
sext               : {m, n} (fin m, m >= n, n >= 1) => [n] -> [m]
ratio              : Integer -> Integer -> Rational
/.                 : {a} (Field a) => a -> a -> a
recip              : {a} (Field a) => a -> a
floor              : {a} (Round a) => a -> Integer
ceiling            : {a} (Round a) => a -> Integer
trunc              : {a} (Round a) => a -> Integer
roundAway          : {a} (Round a) => a -> Integer
roundToEven        : {a} (Round a) => a -> Integer
```

GF(2) polynomial arithmetic

```
pdiv  : {u, v} (fin u, fin v) => [u] -> [v] -> [u]
pmod  : {u, v} (fin u, fin v) => [u] -> [1 + v] -> [v]
pmult : {u, v} (fin u, fin v) => [1 + u] -> [1 + v] -> [1 + u + v]
```

Sequences

```
take      : {front, back, a} (fin front) => [front + back]a -> [front]a
drop      : {front, back, a} (fin front) => [front + back]a -> [back]a
#         : {front, back, a} (fin front) => [front]a -> [back]a -> [front + back]a
join      : {parts, each, a} (fin each) => [parts][each]a -> [parts * each]a
split     : {parts, each, a} (fin each) => [parts * each]a -> [parts][each]a
groupBy   : {each, parts, a} (fin each) => [parts * each]a -> [parts][each]a
transpose : {rows, cols, a} [rows][cols]a -> [cols][rows]a
reverse   : {n, a} (fin n) => [n]a -> [n]a
head      : {n, a} [1 + n]a -> a
tail      : {n, a} [1 + n]a -> [n]a
last      : {n, a} (fin n) => [1 + n]a -> a
```

Indexing, updates

```
@       : {n, a, ix} (Integral ix) => [n]a -> ix -> a
!       : {n, a, ix} (fin n, Integral ix) => [n]a -> ix -> a
@@      : {n, k, ix, a} (Integral ix) => [n]a -> [k]ix -> [k]a
!!      : {n, k, ix, a} (fin n, Integral ix) => [n]a -> [k]ix -> [k]a
update     : {n, a, ix} (fin ix) => [n]a -> [ix] -> a -> [n]a
updateEnd  : {n, a, ix} (fin n, Integral ix) => [n]a -> ix -> a -> [n]a
updates    : {n, k, ix, a} (Integral ix, fin k) => [n]a -> [k]ix -> [k]a -> [n]a
updatesEnd : {n, k, ix, a} (fin n, Integral ix, fin k) => [n]a -> [k]ix -> [k]a -> [n]a
```

Shifting, rotating

```
>>>, <<< : {n, ix, a} (fin n, Integral ix) => [n]a -> ix -> [n]a
>>, <<   : {n, ix, a} (Integral ix, Zero a) => [n]a -> ix -> [n]a
>>$      : {n, ix} (fin n, n >= 1, Integral ix) => [n] -> ix -> [n]
```

Functional programming

```
iterate  : {a} (a -> a) -> a -> [inf]a
repeat   : {n, a} a -> [n]a
map      : {n, a, b} (a -> b) -> [n]a -> [n]b
zip      : {n, a, b} [n]a -> [n]b -> [n](a, b)
zipWith  : {n, a, b, c} (a -> b -> c) -> [n]a -> [n]b -> [n]c
foldl    : {n, a, b} (fin n) => (a -> b -> a) -> a -> [n]b -> a
foldr    : {n, a, b} (fin n) => (a -> b -> b) -> b -> [n]a -> b
scanl    : {n, b, a} (b -> a -> b) -> b -> [n]a -> [1 + n]b
scanr    : {n, a, b} (fin n) => (a -> b -> b) -> b -> [n]a -> [1 + n]b
sum      : {n, a} (fin n, Arith a) => [n]a -> a
and, or  : {n} (fin n) => [n] -> Bit
all, any : {n, a} (fin n) => (a -> Bit) -> [n]a -> Bit
curry    : {a, b, c} ((a, b) -> c) -> a -> b -> c
uncurry  : {a, b, c} (a -> b -> c) -> (a, b) -> c
elem     : {n, a} (fin n, Eq a) => a -> [n]a -> Bit
```


Miscellaneous

```
length : {n, a, b} (fin n, Literal n b) => [n]a -> b
zero   : {a} (Zero a) => a
```

Representing exceptions

```
undefined : {a} a
error     : {a, n} (fin n) => String n -> a
assert    : {a, n} (fin n) => Bit -> String n -> a -> a
trace     : {n, a, b} [n][8] -> a -> b -> b
traceVal  : {n, a} [n][8] -> a -> a
```

Appendix C

Enigma simulator

In this appendix we present the Cryptol code for the enigma machine in its entirety for reference purposes. Chapter 3 has a detailed discussion on how the enigma machine worked, and the construction of the Cryptol model below.

```
/*
 * Cryptol Enigma Simulator
 *
 * Copyright (c) 2010-2018 Galois, Inc.
 * Distributed under the terms of the BSD3 license (see LICENSE file)
 */

// Helper synonyms:
// type Char        = [8]
module Enigma where

type Permutation = String 26

// Enigma components:
type Plugboard   = Permutation
type Rotor       = [26](Char, Bit)
type Reflector   = Permutation

// An enigma machine with n rotors:
type Enigma n    = { plugboard : Plugboard,
                     rotors    : [n]Rotor,
                     reflector : Reflector
                   }

// Check membership in a sequence:
elem : {a, b} (fin 0, fin a, Cmp b) => (b, [a]b) -> Bit
elem (x, xs) = matches ! 0
  where matches = [False] # [ m || (x == e)  | e <- xs
                                             | m <- matches
                            ]
// Inverting a permutation lookup:
private
  invSubst : (Permutation, Char) -> Char
  invSubst (key, c) = candidates ! 0
    where candidates = [0] # [ if c == k then a else p
                             | k <- key
                             | a <- ['A' .. 'Z']
                             | p <- candidates
                             ]

// Constructing a rotor
mkRotor : {n} (fin n) => (Permutation, String n) -> Rotor
mkRotor (perm, notchLocations) = [ (p, elem (p, notchLocations))
                                 | p <- perm
                                 ]

```

```
// Action of a single rotor on a character
// Note that we encrypt and then rotate, if necessary
scramble : (Bit, Char, Rotor) -> (Bit, Char, Rotor)
scramble (rotate, c, rotor) = (notch, c', rotor')
  where
    (c', _)    = rotor @ (c - 'A')
    (_, notch) = rotor @ 0
    rotor'     = if rotate then rotor <<< 1 else rotor

// Connecting rotors in a sequence
joinRotors : {n} (fin n) => ([n]Rotor, Char) -> ([n]Rotor, Char)
joinRotors (rotors, inputChar) = (rotors', outputChar)
  where
    initRotor = mkRotor (['A' .. 'Z'], [])
    ncrs : [n+1](Bit, [8], Rotor)
    ncrs = [(True, inputChar, initRotor)]
                # [ scramble (notch, char, r)
                  | r <- rotors
                  | (notch, char, rotor') <- ncrs
                  ]
    rotors' = tail [ r | (_, _, r) <- ncrs ]
    (_, outputChar, _) = ncrs ! 0

// Following the signal through a single rotor, forward and backward
substFwd, substBwd : (Permutation, Char) -> Char
substFwd (perm, c) = perm @ (c - 'A')
substBwd (perm, c) = invSubst (perm, c)

// Route the signal back from the reflector, chase through rotors
backSignal : {n} (fin n) => ([n]Rotor, Char) -> Char
backSignal (rotors, inputChar) = cs ! 0
  where   cs = [inputChar] # [ substBwd ([ p | (p, _) <- r ], c)
                             | r <- reverse rotors
                             | c <- cs
                             ]

// The full enigma loop, from keyboard to lamps:
// The signal goes through the plugboard, rotors, and the reflector,
// then goes back through the sequence in reverse, out of the
// plugboard and to the lamps
enigmaLoop : {n} (fin n) => (Plugboard, [n]Rotor, Reflector, Char) -> ([n]Rotor, Char)
enigmaLoop (pboard, rotors, refl, c0) = (rotors', c5)
  where
    c1 = substFwd (pboard, c0)
    (rotors', c2) = joinRotors (rotors, c1)
    c3 = substFwd (refl, c2)
    c4 = backSignal(rotors, c3)
    c5 = substBwd (pboard, c4)

// Construct a machine out of parts
mkEnigma : {n} (Plugboard, [n]Rotor, Reflector, [n]Char) -> Enigma n
mkEnigma (pboard, rs, refl, startingPositions) =
    { plugboard  = pboard
    , rotors     = [ r <<< (s - 'A')
                   | r <- rs
                   | s <- startingPositions
                   ]
    , reflector  = refl
    }

// Encryption/Decryption
enigma : {n, m} (fin n, fin m) => (Enigma n, String m) -> String m
enigma (m, pt) = tail [ c | (_, c) <- rcs ]
  where rcs = [(m.rotors, '*')] #
              [ enigmaLoop (m.plugboard, r, m.reflector, c)
              | c      <- pt
              | (r, _) <- rcs
              ]
```



```
// Decryption is the same as encryption:
// dEnigma : {n, m} (fin n, fin m) => (Enigma n, String m) -> String m
dEnigma = enigma


// Build an example enigma machine:
plugboard : Plugboard
plugboard = "HBGDEFCAIJKOWNLPXRSVYTMQUZ"

rotor1, rotor2, rotor3 : Rotor
rotor1 = mkRotor ("RJICAWVQZODLUPYFEHXSMTKNGB", "IO")
rotor2 = mkRotor ("DWYOLETKNVQPHURZJMSFIGXCBA", "B")
rotor3 = mkRotor ("FGKMAJWUOVNRYIZETDPSHBLCQX", "CK")

reflector : Reflector
reflector = "FEIPBATSCYVUWZQDOXHGLKMRJN"

modelEnigma : Enigma 3
modelEnigma = mkEnigma (plugboard, [rotor1, rotor2, rotor3], reflector, "GCR")

/* Example run:

   cryptol> :set ascii=on
   cryptol> enigma  (modelEnigma, "ENIGMAWASAREALLYCOOLMACHINE")
   UPEKTBSDROBVTUJGNCEHHGBXGTF
   cryptol> dEnigma (modelEnigma, "UPEKTBSDROBVTUJGNCEHHGBXGTF")
   ENIGMAWASAREALLYCOOLMACHINE
*/

all: {a, n} (fin n) => (a->Bit) -> [n]a -> Bit
all fn xs = folds ! 0 where
    folds = [True] # [ fn x && p | x <- xs
                                 | p <- folds]
checkReflectorFwdBwd : Reflector -> Bit
checkReflectorFwdBwd refl = all check ['A' .. 'Z']
    where check c = substFwd (refl, c) == substBwd (refl, c)
```

Appendix D

AES in Cryptol

In this appendix we present the Cryptol code for the AES in its entirety for reference purposes. Chapter 5 has a detailed discussion on how AES works, and the construction of the Cryptol model below.

In the below code, simply set `Nk` to be 4 for AES128, 6 for AES192, and 8 for AES256 on line 19. No other modifications are required for obtaining these AES variants. Note that we have rearranged the code given in chapter 5 below for ease of reading.

```
1  // Cryptol AES Implementation
2  // Copyright (c) 2010-2013, Galois Inc.
3  // www.cryptol.net
4  // You can freely use this source code for educational purposes.
5
6  // This is a fairly close implementation of the FIPS-197 standard:
7  //   http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
8
9  // Nk: Number of blocks in the key
10 // Must be one of 4 (AES128), 6 (AES192), or 8 (AES256)
11 // Aside from this line, no other code below needs to change for
12 // implementing AES128, AES192, or AES256
13 module AES where
14
15 type AES128 = 4
16 type AES192 = 6
17 type AES256 = 8
18
19 type Nk = AES128
20
21 // For Cryptol 2.x | x > 0
22 // NkValid: `Nk -> Bit
23 // property NkValid k = (k == `AES128) || (k == `AES192) || (k == `AES256)
24
25 // Number of blocks and Number of rounds
26 type Nb = 4
27 type Nr = 6 + Nk
28
29 type AESKeySize  = (Nk*32)
30
31 // Helper type definitions
32 type GF28        = [8]
33 type State       = [4][Nb]GF28
34 type RoundKey    = State
35 type KeySchedule = (RoundKey, [Nr-1]RoundKey, RoundKey)
36 // GF28 operations
37 gf28Add : {n} (fin n) => [n]GF28 -> GF28
38 gf28Add ps = sums ! 0
39   where sums = [zero] # [  p ^ s | p <- ps | s <- sums ]
40
41 irreducible = <| x^^8 + x^^4 + x^^3 + x + 1 |>
42
43 gf28Mult : (GF28, GF28) -> GF28
```

```
gf28Mult (x, y) = pmod(pmult x y) irreducible

gf28Pow : (GF28, [8]) -> GF28
gf28Pow (n, k) = pow k
  where   sq x  = gf28Mult (x, x)
          odd x = x ! 0
          pow i = if i == 0 then 1
                  else if odd i
                       then gf28Mult(n, sq (pow (i >> 1)))
                       else sq (pow (i >> 1))

gf28Inverse : GF28 -> GF28
gf28Inverse x = gf28Pow (x, 254)

gf28DotProduct : {n} (fin n) => ([n]GF28, [n]GF28) -> GF28
gf28DotProduct (xs, ys) = gf28Add [ gf28Mult (x, y) | x <- xs
                                                    | y <- ys ]

gf28VectorMult : {n, m} (fin n) => ([n]GF28, [m][n]GF28) -> [m]GF28
gf28VectorMult (v, ms) = [ gf28DotProduct(v, m) | m <- ms ]

gf28MatrixMult : {n, m, k} (fin m) => ([n][m]GF28, [m][k]GF28) -> [n][k]GF28
gf28MatrixMult (xss, yss) = [ gf28VectorMult(xs, yss') | xs <- xss ]
   where yss' = transpose yss

// The affine transform and its inverse
xformByte : GF28 -> GF28
xformByte b = gf28Add [b, (b >>> 4), (b >>> 5), (b >>> 6), (b >>> 7), c]
   where c = 0x63

xformByte' : GF28 -> GF28
xformByte' b = gf28Add [(b >>> 2), (b >>> 5), (b >>> 7), d] where d = 0x05
// The SubBytes transform and its inverse
SubByte : GF28 -> GF28
SubByte b = xformByte (gf28Inverse b)

SubByte' : GF28 -> GF28
SubByte' b = sbox@b

SubBytes : State -> State
SubBytes state = [ [ SubByte' b | b <- row ] | row <- state ]


InvSubByte : GF28 -> GF28
InvSubByte b = gf28Inverse (xformByte' b)

InvSubBytes : State -> State
InvSubBytes state = [ [ InvSubByte b | b <- row ] | row <- state ]

// The ShiftRows transform and its inverse
ShiftRows : State -> State
ShiftRows state = [ row <<< shiftAmount | row <- state
                                        | shiftAmount <- [0 .. 3]
                  ]

InvShiftRows : State -> State
InvShiftRows state = [ row >>> shiftAmount | row <- state
                                           | shiftAmount <- [0 .. 3]
                     ]

// The MixColumns transform and its inverse
MixColumns : State -> State
MixColumns state = gf28MatrixMult (m, state)
    where m = [[2, 3, 1, 1],
               [1, 2, 3, 1],
               [1, 1, 2, 3],
               [3, 1, 1, 2]]
```



```
InvMixColumns : State -> State
InvMixColumns state = gf28MatrixMult (m, state)
    where m = [[0x0e, 0x0b, 0x0d, 0x09],
               [0x09, 0x0e, 0x0b, 0x0d],
               [0x0d, 0x09, 0x0e, 0x0b],
               [0x0b, 0x0d, 0x09, 0x0e]]

// The AddRoundKey transform
AddRoundKey : (RoundKey, State) -> State
AddRoundKey (rk, s) = rk ^ s
// Key expansion
Rcon : [8] -> [4]GF28
Rcon i = [(gf28Pow (<| x |>, i-1)), 0, 0, 0]

SubWord : [4]GF28 -> [4]GF28
SubWord bs = [ SubByte' b | b <- bs ]

RotWord : [4]GF28 -> [4]GF28
RotWord [a0, a1, a2, a3] = [a1, a2, a3, a0]

NextWord : ([8],[4][8],[4][8]) -> [4][8]
NextWord(i, prev, old) = old ^ mask
   where mask = if i % `Nk == 0
                then SubWord(RotWord(prev)) ^ Rcon (i / `Nk)
                else if (`Nk > 6) && (i % `Nk == 4)
                     then SubWord(prev)
                     else prev


ExpandKeyForever : [Nk][4][8] -> [inf]RoundKey
ExpandKeyForever seed = [ transpose g | g <- groupBy`{4} (keyWS seed) ]

keyWS : [Nk][4][8] -> [inf][4][8]
keyWS seed    = xs
     where xs = seed # [ NextWord(i, prev, old)
                       | i    <- [ `Nk ... ]
                       | prev <- drop`{Nk-1} xs
                       | old  <- xs
                       ]

ExpandKey : [AESKeySize] -> KeySchedule
ExpandKey key = (keys @ 0, keys @@ [1 .. (Nr - 1)], keys @ `Nr)
  where   seed : [Nk][4][8]
          seed = split (split key)
          keys = ExpandKeyForever seed

fromKS : KeySchedule -> [Nr+1][4][32]
fromKS (f, ms, l) = [ formKeyWords (transpose k) | k <- [f] # ms # [l] ]
    where formKeyWords bbs = [ join bs | bs <- bbs ]

// AES rounds and inverses
AESRound : (RoundKey, State) -> State
AESRound (rk, s) = AddRoundKey (rk, MixColumns (ShiftRows (SubBytes s)))

AESFinalRound : (RoundKey, State) -> State
AESFinalRound (rk, s) = AddRoundKey (rk, ShiftRows (SubBytes s))

AESInvRound : (RoundKey, State) -> State
AESInvRound (rk, s) =
      InvMixColumns (AddRoundKey (rk, InvSubBytes (InvShiftRows s)))
AESFinalInvRound : (RoundKey, State) -> State
AESFinalInvRound (rk, s) = AddRoundKey (rk, InvSubBytes (InvShiftRows s))

// Converting a 128 bit message to a State and back
msgToState : [128] -> State
msgToState msg = transpose (split (split msg))

stateToMsg : State -> [128]
```

```
stateToMsg st = join (join (transpose st))

// AES Encryption
aesEncrypt : ([128], [AESKeySize]) -> [128]
aesEncrypt (pt, key) = stateToMsg (AESFinalRound (kFinal, rounds ! 0))
  where   (kInit, ks, kFinal) = ExpandKey key
          state0 = AddRoundKey(kInit, msgToState pt)
          rounds = [state0] # [ AESRound (rk, s) | rk <- ks
                                                 | s <- rounds
                              ]

// AES Decryption
aesDecrypt : ([128], [AESKeySize]) -> [128]
aesDecrypt (ct, key) = stateToMsg (AESFinalInvRound (kFinal, rounds ! 0))
  where   (kFinal, ks, kInit) = ExpandKey key
          state0 = AddRoundKey(kInit, msgToState ct)
          rounds = [state0] # [ AESInvRound (rk, s)
                              | rk <- reverse ks
                              | s  <- rounds
                              ]

sbox : [256]GF28
sbox = [
 0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67,
 0x2b, 0xfe, 0xd7, 0xab, 0x76, 0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59,
 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, 0xb7,
 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1,
 0x71, 0xd8, 0x31, 0x15, 0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05,
 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, 0x09, 0x83,
 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29,
 0xe3, 0x2f, 0x84, 0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b,
 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, 0xd0, 0xef, 0xaa,
 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c,
 0x9f, 0xa8, 0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc,
 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, 0xcd, 0x0c, 0x13, 0xec,
 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19,
 0x73, 0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee,
 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, 0xe0, 0x32, 0x3a, 0x0a, 0x49,
 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
 0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4,
 0xea, 0x65, 0x7a, 0xae, 0x08, 0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6,
 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, 0x70,
 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9,
 0x86, 0xc1, 0x1d, 0x9e, 0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e,
 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, 0x8c, 0xa1,
 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0,
 0x54, 0xbb, 0x16]

// Test runs:

// cryptol> aesEncrypt (0x3243f6a8885a308d313198a2e0370734,   \
//                      0x2b7e151628aed2a6abf7158809cf4f3c)
// 0x3925841d02dc09fbdc118597196a0b32
// cryptol> aesEncrypt (0x00112233445566778899aabbccddeeff,   \
//                      0x000102030405060708090a0b0c0d0e0f)
// 0x69c4e0d86a7b0430d8cdb78070b4c55a
property AESCorrect msg key = aesDecrypt (aesEncrypt (msg, key), key) == msg
```


Appendix E

Technicalities

The summary below describes language features, as well as commands that are available at the `Cryptol>` prompt. Commands all begin with the `:` character.

E.1 Language features

The Cryptol language is a size-polymorphic dependently-typed programming language with support for polymorphic recursive functions. It has a small syntax tuned for applied cryptography, a lightweight module system, a Read–Eval–Print loop (REPL) top-level, and a rich set of built-in tools for performing high-assurance (literate) programming. Cryptol performs fairly advanced type inference, though as with most mainstream strongly typed functional languages, types can be manually specified as well. What follows is a brief tour of Cryptol's most salient language features.

Case sensitivity Cryptol identifiers are case sensitive. `A` and `a` are two different things.

Indentation and whitespace Cryptol uses indentation level (instead of `{}`) to denote blocks. Whitespace within a line is immaterial, as is the specific amount of indentation. However, consistent indentation will save you tons of trouble down the road! Do not mix tabs and spaces for your indentation. Spaces are generally preferred.

Escape characters Long lines can be continued with the end-of-line escape character `\`, as in many programming languages. There are no built-in character escape characters, as Cryptol performs no interpretation on bytes beyond printing byte streams out in ASCII, as discussed above.

Comments Block comments are enclosed in `/*` and `*/`, and they can be nested. Line comments start with `//` and run to the end of the line.

Order of definitions The order of definitions is immaterial. You can write your definitions in any order, and earlier entries can refer to later ones.

Typing Cryptol is strongly typed. This means that the interpreter will catch most common mistakes in programming during the type-checking phase, before runtime.

Type inference Cryptol has type inference. This means that the user can omit type signatures because the inference engine will supply them.

Type signatures While explicit type signatures are optional, writing them down is considered good practice.

Polymorphism Cryptol functions can be polymorphic, which means they can operate on many different types. Be aware that the type that Cryptol infers might be too polymorphic, so it is good practice to write your signatures, or at least check that what Cryptol inferred is what you had in mind.

Module system Each Cryptol file defines a *module*. Modules allow Cryptol developers to manage which definitions are exported (the default behavior) and which definitions are internal-only (*private*). At the beginning of each Cryptol file, you specify its name and use `import` to specify the modules on which it relies. Definitions are `public` by default, but you can hide them from modules that import your code via the `private` keyword at the start of each private definition, like this:

```
module test where
private
  hiddenConst = 0x5      // hidden from importing modules
// end of indented block indicates symbols are available to importing modules
revealedConst = 0x15
```

Note that the filename should correspond to the module name, so `module test` must be defined in a file called `test.cry`.

Literate programming You can feed LaTeX files to Cryptol (i.e., files with extension `.tex`). Cryptol will look for `\begin{code}` and `\end{code}` marks to extract Cryptol code. Everything else will be comments as far as Cryptol is concerned. In fact, the book you are reading is a literate Cryptol program.

Completion On UNIX-based machines, you can press tab at any time and Cryptol will suggest completions based on the context. You can retrieve your prior commands using the usual means (arrow keys or Emacs keybindings).

E.2 Commands

Querying types You can ask Cryptol to tell you the type of an expression by typing `:type <expr>` (or `:t` for short). If `foo` is the name of a definition (function or otherwise), you can ask its type by issuing `:type foo`. It is common practice to define a function, ask Cryptol its type, and copy the response back to your source code. While this is somewhat contrived, it is usually better than not writing signatures at all. In order to query the type of an infix operator (e.g., `+`, `==`, etc.) you will need to surround the operator with `()`, like this:

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

Browsing definitions The command `:browse` (or `:b` for short) will display all the names you have defined, along with their types. Using `:browse` with the name of a loaded module will show the declarations from that module only.

Getting help The command `:help` will show you all the available commands. Using `:help` with the name of a defined function or type synonym will display its corresponding help text. Other useful implicit help invocations are: (a) to type tab at the `Cryptol>` prompt, which will list all of the operators available in Cryptol code, (b) typing `:set` with no argument, which shows you the parameters that can be set, and (c), as noted elsewhere, `:browse` to see the names of functions and type aliases you have defined, along with their types.

Option	Default value	Meaning
`ascii`	`off`	print sequences of bytes as a string
`base`	`16`	numeric base for printing words
`ignore-safety`	`off`	ignore safety predicates when performing `:sat` or `:prove` checks
`infLength`	`5`	number of elements to show from an infinite sequence
`prover`	`z3`	which SMT solver to use for `:prove`
`satNum`	`1`	maximum number of solutions to show for `:sat`
`show-examples`	`on`	whether to print counterexamples and models
`smtfile`	`-`	file to write SMT problems when `prover = offline`
`tests`	`100`	number of tests to run for `:check`

Environment options A variety of environment options are set through the use of the `:set` command. These options may change over time and some options may be available only on specific platforms. The most useful configuration options are summarized in section E.2. Help information about all flags can found by typing `:help :set flag`.

Quitting You can quit Cryptol by using the command `:quit` (aka `:q`). On Mac/Linux you can press Ctrl-D, and on Windows use Ctrl-Z, for the same effect.

Loading and reloading files You load your program in Cryptol using `:load <filename>` (or `:l` for short). However, it is customary to use the extension `.cry` for Cryptol programs. If you edit the source file loaded into Cryptol from a separate context, you can reload it into Cryptol using the command `:reload` (abbreviated `:r`).

Invoking your editor You can invoke your editor using the command `:edit` (abbreviated `:e`). The default editor invoked is `vi`. You override the default using the standard `EDITOR` environmental variable in your shell.

Running shell commands You can run Unix shell commands from within Cryptol like this: `:! cat test.cry`.

Changing working directory You can change the current working directory of Cryptol like this: `:cd some/path`. Note that the path syntax is platform-dependent.

Loading a module At the Cryptol prompt you can load a module by name with the `:module` command.

The next commands all operate on *properties*. All take either one or zero arguments. If one argument is provided, then that property is the focus of the command; otherwise all properties in the current context are checked. All three commands are covered in detail in chapter 4.

Checking a property through random testing The `:check` command performs random value testing on a property to increase one's confidence that the property is valid. See section 4.3 for more detailed information.

Checking a property through exhaustive testing The `:exhaust` command enumerates all the possible input values of a property to test that it is correct for every possible value. This is infeasible except for properties with quite small input domains!

Verifying a property through automated theorem proving The `:prove` command uses an external SMT solver to attempt to automatically formally prove that a given property is valid. See subsection 4.2.1 for more detailed information.

Finding a satisfying assignment for a property The `:sat` command uses an external SMT solver to attempt to find a satisfying assignment to a property. See section 4.4 for more detailed information.

Proving that a property is safe The `:safe` command uses an external SMT solver to attempt to prove that the given term is safe for all inputs, which means it cannot encounter a run-time error. Unlike the other commands in this section, `:safe` *must* be given an argument and can be called on values that compute types other than `Bit`. Most types, except infinite streams and functions, are allowed in the result type.

Appendix F

Cryptol Syntax

F.1 Layout

Groups of declarations are organized based on indentation. Declarations with the same indentation belong to the same group. Lines of text that are indented more than the beginning of a declaration belong to that declaration, while lines of text that are indented less terminate a group of declarations. Groups of declarations appear at the top level of a Cryptol file, and inside `where` blocks in expressions. For example, consider the following declaration group:

```
f x = x + y + z
  where
  y = x * x
  z = x + y

g y = y
```

This group has two declarations, one for `f` and one for `g`. All the lines between `f` and `g` that are indented more than `f` belong to `f`.

This example also illustrates how groups of declarations may be nested within each other. For example, the `where` expression in the definition of `f` starts another group of declarations, containing `y` and `z`. This group ends just before `g`, because `g` is indented less than `y` and `z`.

F.2 Comments

Cryptol supports block comments, which start with `/*` and end with `*/`, and line comments, which start with `//` and terminate at the end of the line. Block comments may be nested arbitrarily.

Examples:

```
/* This is a block comment */
// This is a line comment
/* This is a /* Nested */ block comment */
```

F.3 Identifiers

Cryptol identifiers consist of one or more characters. The first character must be either an English letter or underscore (`_`). The following characters may be an English letter, a decimal digit, underscore (`_`), or a prime (`'`). Some identifiers have special meaning in the language, so they may not be used in programmer-defined names (see Keywords).

Examples:

```
name    name1    name'    longer_name
Name    Name2    Name''   longerName
```

F.4 Keywords and Built-in Operators

The following identifiers have special meanings in Cryptol, and may not be used for programmer defined names:

`else`	`include`	`property`	`let`	`newtype`	`primitive`
`extern`	`module`	`then`	`import`	`infixl`	`parameter`
`if`	`newtype`	`type`	`as`	`infixr`	`constraint`
`private`	`pragma`	`where`	`hiding`	`infix`	

The following table contains Cryptol's operators and their associativity with lowest precedence operators first, and highest precedence last.

Table F.1: Operator precedences.

Operator	Associativity
`==>`	right
`\/`	right
`/\`	right
`== != === !==`	not associative
`> < <= >= <$ >$ <=$ >=$`	not associative
`||`	right
`^`	left
`&&`	right
`#`	right
`>> << >>> <<< >>$`	left
`+ -`	left
`* / % /$ %$`	left
`^^`	right
`@ @@ ! !!`	left
(unary) `- ~`	right

F.5 Built-in Type-level Operators

Cryptol includes a variety of operators that allow computations on the numeric types used to specify the sizes of sequences.

Table F.2: Type-level operators

Operator	Meaning
`+`	Size addition
`-`	Size subtraction
`*`	Size multiplication
`/`	Size division
`/^`	Size ceiling division (`/` rounded up)
`%`	Size modulus
`%^`	Size ceiling modulus (`%` rounded up)
`^^`	Size exponentiation
`lg2`	Size logarithm (base 2)
`width`	Size width (`lg2` rounded up)
`max`	Size maximum
`min`	Size minimum

F.6 Numeric Literals

Numeric literals may be written in binary, octal, decimal, or hexadecimal notation. The base of a literal is determined by its prefix: `0b` for binary, `0o` for octal, no special prefix for decimal, and `0x` for hexadecimal.

Examples:

```
254                 // Decimal literal
0254                // Decimal literal
0b11111110          // Binary literal
0o376               // Octal literal
0xFE                // Hexadecimal literal
0xfe                // Hexadecimal literal
```

Numeric literals in binary, octal, or hexadecimal notation result in bit sequences of a fixed length (i.e., they have type `[n]` for some `n`). The length is determined by the base and the number of digits in the literal. Decimal literals are overloaded, and so the type is inferred from context in which the literal is used. Examples:

```
0b1010              // : [4],   1 * number of digits
0o1234              // : [12],  3 * number of digits
0x1234              // : [16],  4 * number of digits

10                  // : {a}. (Literal 10 a) => a
                    // a = Integer or [n] where n >= width 10
```

Numeric literals may also be written as polynomials by writing a polynomial expression in terms of `x` between an opening `<|` and a closing `|>`. Numeric literals in polynomial notation result in bit sequences of length one more than the degree of the polynomial. Examples:

```
<| x^^6 + x^^4 + x^^2 + x^^1 + 1 |>  // : [7], equal to 0b1010111
<| x^^4 + x^^3 + x |>                // : [5], equal to 0b11010
```

Cryptol also supports fractional literals using binary (prefix `0b`), octal (prefix `0o`), decimal (no prefix), and hexadecimal (prefix `ox`) digits. A fractional literal must contain a `.` and may optionally have an exponent. The base of the exponent for binary, octal, and hexadecimal literals is 2 and the exponent is marked using the symbol `p`. Decimal fractional literals use exponent base 10, and the symbol `e`. Examples:

```
10.2
10.2e3            // 10.2 * 10^3
0x30.1            // 3 * 64 + 1/16
0x30.1p4          // (3 * 64 + 1/16) * 2^4
```

All fractional literals are overloaded and may be used with types that support fractional numbers (e.g., `Rational`, and the `Float` family of types).

Some types (e.g. the `Float` family) cannot represent all fractional literals precisely. Such literals are rejected statically when using binary, octal, or hexadecimal notation. When using decimal notation, the literal is rounded to the closes represental even number.

All numeric literals may also include `_`, which has no effect on the literal value but may be used to improve readability. Here are some examples:

```
0b_0000_0010
0x_FFFF_FFEA
```

F.7 Expressions

This section provides an overview of the Cryptol's expression syntax.

Calling Functions

```
f 2             // call `f` with parameter `2`
g x y           // call `g` with two parameters: `x` and `y`
h (x,y)         // call `h` with one parameter,  the pair `(x,y)`
```

Prefix Operators

```
-2              // call unary `-` with parameter `2`
- 2             // call unary `-` with parameter `2`
f (-2)          // call `f` with one argument: `-2`,  parens are important
-f 2            // call unary `-` with parameter `f 2`
- f 2           // call unary `-` with parameter `f 2`
```

Infix Functions

```
2 + 3           // call `+` with two parameters: `2` and `3`
2 + 3 * 5       // call `+` with two parameters: `2` and `3 * 5`
(+) 2 3         // call `+` with two parameters: `2` and `3`
f 2 + g 3       // call `+` with two parameters: `f 2` and `g 3`
- 2 + - 3       // call `+` with two parameters: `-2` and `-3`
- f 2 + - g 3
```

Type Annotations

```
x : Bit         // specify that `x` has type `Bit`
f x : Bit       // specify that `f x` has type `Bit`
- f x : [8]     // specify that `- f x` has type `[8]`
2 + 3 : [8]     // specify that `2 + 3` has type `[8]`
\x -> x : [8]   // type annotation is on `x`, not the function
if x
  then y
  else z : Bit  // the type annotation is on `z`, not the whole `if`
[1..9 : [8]]    // specify that elements in `[1..9]` have type `[8]`
```

Local Declarations

Local declarations have the weakest precedence of all expressions.

```
2 + x : [T]
  where
  type T = 8
  x      = 2          // `T` and `x` are in scope of `2 + x : `[T]`

if x then 1 else 2
  where x = 2         // `x` is in scope in the whole `if`

\y -> x + y
  where x = 2         // `y` is not in scope in the defintion of `x`
```

Block Arguments

When used as the last argument to a function call, `if` and lambda expressions do not need parens:

```
f \x -> x       // call `f` with one argument `x -> x`
2 + if x
      then y
      else z    // call `+` with two arguments: `2` and `if ...`
```


F.8 Bits

The type `Bit` has two inhabitants: `True` and `False`. These values may be combined using various logical operators, or constructed as results of comparisons.

Table F.3: Bit operations.

Operator	Associativity	Description
`==>`	right	Short-cut implication
`\/`	right	Short-cut or
`/\`	right	Short-cut and
`!= ==`	none	Not equals, equals
`> < <= >= <$ >$ <=$ >=$`	none	Comparisons
`||`	right	Logical or
`^`	left	Exclusive-or
`&&`	right	Logical and
`~`	right	Logical negation

F.9 Multi-way Conditionals

The `if ... then ... else` construct can be used with multiple branches. For example:

```
x = if y % 2 == 0 then 22 else 33

x = if y % 2 == 0 then 1
     | y % 3 == 0 then 2
     | y % 5 == 0 then 3
     else 7
```

F.10 Tuples and Records

Tuples and records are used for packaging multiple values together. Tuples are enclosed in parentheses, while records are enclosed in curly braces. The components of both tuples and records are separated by commas. The components of tuples are expressions, while the components of records are a label and a value separated by an equal sign. Examples:

```
(1,2,3)           // A tuple with 3 component
()                // A tuple with no components

{ x = 1, y = 2 }  // A record with two fields, `x` and `y`
{}                // A record with no fields
```

The components of tuples are identified by position, while the components of records are identified by their label, and so the ordering of record components is not important. Examples:

```
           (1,2) == (1,2)               // True
           (1,2) == (2,1)               // False

{ x = 1, y = 2 } == { x = 1, y = 2 }    // True
{ x = 1, y = 2 } == { y = 2, x = 1 }    // True
```

F.10.1 Accessing Fields

The components of a record or a tuple may be accessed in two ways: via pattern matching or by using explicit component selectors. Explicit component selectors are written as follows:

```
(15, 20).0           == 15
(15, 20).1           == 20

{ x = 15, y = 20 }.x == 15
```

Explicit record selectors may be used only if the program contains sufficient type information to determine the shape of the tuple or record. For example:

```
type T = { sign : Bit, number : [15] }

// Valid definition:
// the type of the record is known.
isPositive : T -> Bit
isPositive x = x.sign

// Invalid definition:
// insufficient type information.
badDef x = x.f
```

The components of a tuple or a record may also be accessed using pattern matching. Patterns for tuples and records mirror the syntax for constructing values: tuple patterns use parentheses, while record patterns use braces. Examples:

```
getFst (x,_) = x

distance2 { x = xPos, y = yPos } = xPos ^^ 2 + yPos ^^ 2

f p = x + y where
    (x, y) = p
```

Selectors are also lifted through sequence and function types, point-wise, so that the following equations should hold:

```
xs.l == [ x.l | x <- xs ]     // sequences
f.l  == \x -> (f x).l         // functions
```

Thus, if we have a sequence of tuples, `xs`, then we can quickly obtain a sequence with only the tuples' first components by writing `xs.0`.

Similarly, if we have a function, `f`, that computes a tuple of results, then we can write `f.0` to get a function that computes only the first entry in the tuple.

This behavior is quite handy when examining complex data at the REPL.

F.10.2 Updating Fields

The components of a record or a tuple may be updated using the following notation:

```
// Example values
r = { x = 15, y = 20 }      // a record
t = (True,True)             // a tuple
n = { pt = r, size = 100 }  // nested record

// Setting fields
{ r | x = 30 }          == { x = 30, y = 20 }
{ t | 0 = False }       == (False,True)

// Update relative to the old value
{ r | x -> x + 5 }      == { x = 20, y = 20 }
```



```
// Update a nested field
{ n | pt.x = 10 }       == { pt = { x = 10, y = 20 }, size = 100 }
{ n | pt.x -> x + 10 }  == { pt = { x = 25, y = 20 }, size = 100 }
```

F.11 Sequences

A sequence is a fixed-length collection of elements of the same type. The type of a finite sequence of length `n`, with elements of type `a` is `[n] a`. Often, a finite sequence of bits, `[n] Bit`, is called a *word*. We may abbreviate the type `[n] Bit` as `[n]`. An infinite sequence with elements of type `a` has type `[inf] a`, and `[inf]` is an infinite stream of bits.

```
[e1,e2,e3]                    // A sequence with three elements

[t1 .. t3 ]                   // Sequence enumerations
[t1, t2 .. t3 ]               // Step by t2 - t1
[e1 ... ]                     // Infinite sequence starting at e1
[e1, e2 ... ]                 // Infinite sequence stepping by e2-e1

[ e | p11 <- e11, p12 <- e12  // Sequence comprehensions
    | p21 <- e21, p22 <- e22 ]

x = generate (\i -> e)        // Sequence from generating function
x @ i = e                     // Sequence with index binding
arr @ i @ j = e               // Two-dimensional sequence
```

Note: the bounds in finite sequences (those with `..`) are type expressions, while the bounds in infinite sequences are value expressions.

Table F.4: Sequence operations.

Operator	Description
`#`	Sequence concatenation
`>> <<`	Shift (right, left)
`>>> <<<`	Rotate (right, left)
`>>$`	Arithmetic right shift (on bitvectors only)
`@ !`	Access elements (front, back)
`@@ !!`	Access sub-sequence (front, back)
`update updateEnd`	Update the value of a sequence at a location (front, back)
`updates updatesEnd`	Update multiple values of a sequence (front, back)

There are also lifted pointwise operations.

```
[p1, p2, p3, p4]          // Sequence pattern
p1 # p2                   // Split sequence pattern
```

F.12 Functions

```
\p1 p2 -> e              // Lambda expression
f p1 p2 = e              // Function definition
```

F.13 Local Declarations

```
e where ds
```

Note that by default, any local declarations without type signatures are monomorphized. If you need a local declaration to be polymorphic, use an explicit type signature.

F.14 Explicit Type Instantiation

If `f` is a polymorphic value with type:

```
f : { tyParam } tyParam
f = zero
```

you can evaluate `f`, passing it a type parameter:

```
f `{ tyParam = 13 }
```

F.15 Demoting Numeric Types to Values

The value corresponding to a numeric type may be accessed using the following notation:

```
`t
```

Here `t` should be a type expression with numeric kind. The resulting expression is a finite word, which is sufficiently large to accommodate the value of the type:

```
`t : {n} (fin n, n >= width t) => [n]
```

F.16 Explicit Type Annotations

Explicit type annotations may be added on expressions, patterns, and in argument definitions.

```
e : t

p : t

f (x : t) = ...
```

F.17 Type Signatures

```
f,g : {a,b} (fin a) => [a] b
```

F.18 Type Synonym Declarations

```
type T a b = [a] b
```

F.19 Modules

A ***module*** is used to group some related definitions. Each file may contain at most one module.

```
module M where

type T = [8]

f : [8]
f = 10
```


F.20 Hierarchical Module Names

Module may have either simple or ***hierarchical*** names. Hierarchical names are constructed by gluing together ordinary identifiers using the symbol `::`.

```
module Hash::SHA256 where

sha256 = ...
```

The structure in the name may be used to group together related modules. Also, the Cryptol implementation uses the structure of the name to locate the file containing the definition of the module. For example, when searching for module `Hash::SHA256`, Cryptol will look for a file named `SHA256.cry` in a directory called `Hash`, contained in one of the directories specified by `CRYPTOLPATH`.

F.21 Module Imports

To use the definitions from one module in another module, we use `import` declarations:

```
// Provide some definitions
module M where

f : [8]
f = 2


// Uses definitions from `M`
module N where

import M  // import all definitions from `M`

g = f   // `f` was imported from `M`
```

F.22 Import Lists

Sometimes, we may want to import only some of the definitions from a module. To do so, we use an import declaration with an ***import list***.

```
module M where

f = 0x02
g = 0x03
h = 0x04

module N where

import M(f,g)  // Imports only `f` and `g`, but not `h`

x = f + g
```

Using explicit import lists helps reduce name collisions. It also tends to make code easier to understand, because it makes it easy to see the source of definitions.

F.23 Hiding Imports

Sometimes a module may provide many definitions, and we want to use most of them but with a few exceptions (e.g., because those would result to a name clash). In such situations it is convenient to use a ***hiding*** import:

module M where

```
f = 0x02
g = 0x03
h = 0x04
```

```
module N where

import M hiding (h) // Import everything but `h`

x = f + g
```

F.24 Qualified Module Imports

Another way to avoid name collisions is by using a ***qualified*** import.

```
module M where

f : [8]
f = 2

module N where

import M as P

g = P::f
// `f` was imported from `M`
// but when used it needs to be prefixed by the qualified `P`
```

Qualified imports make it possible to work with definitions that happen to have the same name but are defined in different modules.

Qualified imports may be combined with import lists or hiding clauses:

```
import A as B (f)         // introduces B::f
import X as Y hiding (f)  // introduces everything but `f` from X
                          // using the prefix `X`
```

It is also possible the use the same qualifier prefix for imports from different modules. For example:

```
import A as B
import X as B
```

Such declarations will introduces all definitions from `A` and `X` but to use them, you would have to qualify using the prefix `B:::`.

F.25 Private Blocks

In some cases, definitions in a module might use helper functions that are not intended to be used outside the module. It is good practice to place such declarations in ***private blocks***:

```
module M where

f : [8]
f = 0x01 + helper1 + helper2

private

  helper1 : [8]
  helper1 = 2

  helper2 : [8]
  helper2 = 3
```

The keyword `private` introduce a new layout scope, and all declarations in the block are considered to be private to the module. A single module may contain multiple private blocks. For example, the following module is equivalent to the previous one:

```
module M where

f : [8]
f = 0x01 + helper1 + helper2

private
  helper1 : [8]
  helper1 = 2

private
  helper2 : [8]
  helper2 = 3
```

F.26 Parameterized Modules

```
module M where

parameter
  type n : #              // `n` is a numeric type parameter

  type constraint (fin n, n >= 1)
    // Assumptions about the parameter

  x : [n]                 // A value parameter

// This definition uses the parameters.
f : [n]
f = 1 + x
```

F.27 Named Module Instantiations

One way to use a parameterized module is through a named instantiation:

```
// A parameterized module
module M where

parameter
```

```
  type n : #
  x      : [n]
  y      : [n]

f : [n]
f = x + y


// A module instantiation
module N = M where

type n = 32
x      = 11
y      = helper

helper = 12
```

The second module, N, is computed by instantiating the parameterized module M. Module N will provide the exact same definitions as M, except that the parameters will be replaced by the values in the body of N. In this example, N provides just a single definition, f.

Note that the only purpose of the body of N (the declarations after the `where` keyword) is to define the parameters for M.

F.28 Parameterized Instantiations

It is possible for a module instantiations to be itself parameterized. This could be useful if we need to define some of a module's parameters but not others.

```
// A parameterized module
module M where

parameter
  type n : #
  x      : [n]
  y      : [n]

f : [n]
f = x + y


// A parameterized instantiation
module N = M where

parameter
  x : [32]

type n = 32
y      = helper

helper = 12
```

In this case N has a single parameter x. The result of instantiating N would result in instantiating M using the value of x and 12 for the value of y.

F.29 Importing Parameterized Modules

It is also possible to import a parameterized module without using a module instantiation:

```
module M where

parameter
  x : [8]
  y : [8]

f : [8]
f = x + y


module N where

import `M

g = f { x = 2, y = 3 }
```

A ***backtick*** at the start of the name of an imported module indicates that we are importing a parameterized module. In this case, Cryptol will import all definitions from the module as usual, however every definition will have some additional parameters corresponding to the parameters of a module. All value parameters are grouped in a record.

This is why in the example `f` is applied to a record of values, even though its definition in `M` does not look like a function.

Glossary

AES The Advanced Encryption Standard [12], 49

Cipherkey The key used in a particular encryption/decryption task, 27

Ciphertext The result of encrypting a plaintext message, "unreadable" unless the key is known, 27

Fibonacci numbers The sequence $0, 1, 1, 2, 3, 5, \ldots$ After the elements 0 and 1, each consecutive element is the sum of the two previous numbers [18], 21

NIST National Institute of Standards and Technology. The institution in charge of standardizing cryptoalgorithms (amongst many other things) in USA., 49

Plaintext A "readable" message that we would like to encrypt, the message in the clear, 27

SAT Solver An automated tool for solving boolean satisfiability problems. Cryptol uses SAT/SMT solvers in order to provide its high-assurance capabilities, 44, 46, 48

SMT Solver Satisfiability Modulo Theories: An automated tool for establishing satisfiability problems with respect to certain theories. One of the theories of interest to Cryptol is that of bit-vectors, as it provides a natural medium for translating Cryptol's bit-precise theorems, 44, 46, 48

Bibliography

(Each entry is followed by a list of page numbers on which the citation appears. All cited URLs, unless otherwise stated, were last accessed in November 2010.)

[1] Richard Bird. *Introduction to Functional Programming using Haskell*. Printice Hall Europe, London, second edition, 1998. 20

[2] Andy Carlson. Simulating the enigma cypher machine. `http://homepages.tesco.net/~andycarlson/enigma/simulating_enigma.html`, see the section “Wheel Turnover and The Anomaly”. 35, 84

[3] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. of International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 2000. 46

[4] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002. 49

[5] Levent Erkök, Magnus Carlsson, and Adam Wick. Hardware/software co-verification of cryptographic algorithms using Cryptol. In *Formal Methods in Computer Aided Design, FMCAD'09, Austin, TX, USA*, pages 188–191. IEEE, November 2009. 13

[6] Levent Erkök and John Matthews. Pragmatic equivalence and safety checking in Cryptol. In *Programming Languages meets Program Verification, PLPV'09, Savannah, Georgia, USA*, pages 73–81. ACM Press, January 2009. 13, 16, 43, 44

[7] J. Roger Hindley. *Basic Simple Type Theory*, volume 42. Cambridge University Press, Cambridge, UK, 1997. 13

[8] Brian W. Kernighan and Dennis M. Richie. *The C Programming Language*. Prentice Hall, second edition, 1998. 22

[9] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008. 90

[10] J. R. Lewis and B. Martin. Cryptol: high assurance, retargetable crypto development and validation. In *Military Communications Conference 2003*, volume 2, pages 820–825. IEEE, October 2003. 16

[11] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997. 14

[12] National Institute of Standards and Technology, NIST. Announcing the AES. `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`, November 2001. FIPS Publication 197. 49, 51, 52, 53, 54, 55, 57, 58, 59, 60, 125

[13] Simon L. Peyton Jones and John Hughes. (Editors.) Report on the programming language Haskell 98, a non-strict purely-functional programming language. URL: `www.haskell.org/onlinereport`, February 1999. 14, 18, 22

[14] Simon Singh. *The code book: the evolution of secrecy from Mary, Queen of Scots, to quantum cryptography*. Doubleday, New York, NY, USA, 1999. 27, 33, 36

[15] CVC4 web site. `http://cvc4.cs.nyu.edu/web/`. 44

[16] Wikipedia. Enigma machine — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Enigma_machine&oldid=392040616, 2010. [Online; accessed 21-October-2010]. 33

[17] Wikipedia. Enigma rotor details — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Enigma_rotor_details&oldid=389862975, 2010. [Online; accessed 25-October-2010]. 39

[18] Wikipedia. Fibonacci number — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Fibonacci_number&oldid=390711214, 2010. [Online; accessed 15-October-2010]. 21, 125

[19] Wikipedia. Finite field arithmetic — wikipedia, the free encyclopedia, 2010. [Online; accessed 10-November-2010]. 50

[20] Wikipedia. Polynomial long division — wikipedia, the free encyclopedia, 2010. [Online; accessed 10-November-2010]. 51

[21] Wikipedia. Scytale — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Scytale&oldid=391405769, 2010. [Online; accessed 19-October-2010]. 30

[22] Wikipedia. Substitution cipher — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Substitution_cipher&oldid=389116332, 2010. [Online; accessed 20-October-2010]. 29

[23] Wikipedia. Vigenere cipher — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Vigenere_cipher&oldid=389811286, 2010. [Online; accessed 19-October-2010]. 28

[24] Yices web site. http://yices.csl.sri.com/. 44

Index

GF(2^8), *see* galois field
λ-expression, 18, 45, 47, 86

`&&`, (and), 1
`#`, (append), 6, 7, 42, 67, 84
`~`, (complement), 1, 10, 43
`/`, (divide), 11, 45, 70
`==`, (equal), 11, 12, 70, 74
`^^`, (exponentiate), 11, 41
`>`, (greater than), 12
`>=`, (greater or equal), 12
`<`, (less than), 12
`<=`, (less or equal), 12
`-`, (subtract), 11, 41, 74
`%`, (modulus), 11, 45, 70
`||`, (or), 1, 75
`+`, (add), 11, 12, 16, 41
`!`, (reverse select), 6, 19, 20, 36, 58, 74, 79
`!!`, (reverse permutation), 6
`<<<`, (rotate left), 8, 9, 28, 35, 53, 55, 77, 78
`>>>`, (rotate right), 8, 9, 77, 78
`@`, (select), 6, 9, 29, 35, 56
`<<`, (shift left), 8, 9
`>>`, (shift right), 8, 9
`*`, (multiply), 11, 41, 45
`-`, (unary minus), 11, 12
`_`, (underscore), 35
`^`, (xor), 1, 50, 57
`!=`, (not-equal), 11, 12, 70
AES, 49–58, 60, 103
    `InvMixColumns`, 59
    `InvShiftRows`, 59
    `InvSubBytes`, 59
    `MixColumns`, 54, 59
    `ShiftRows`, 53, 59
    state, 50
    `SubBytes`, 51, 59
`all`, 18, 46, 73, 83, 84, 86
ambiguous constraints, *see* type, ambiguous
`any`, 18, 74
atbash, 29, 79

bit, 1, 13

Caesar's cipher, 27, 28, 46, 79
case sensitivity, 2, 107
characters, 9
cipherkey, 27, 30
ciphertext, 27, 30
command line
    completion, 108
    line continuation, 65, 68, 107
commands
    `:b` (browse), 108
    `:check`, 46, 47, 54, 61, 89
    `:e` (edit), 109
    `:?` (help), 108
    `:l` (load), 16, 109
    `:m` (module load), 109
    `:prove`, 44–47, 89
    `:q` (quit), 109
    `:r` (reload), 16, 109
    `:sat`, 47, 48
    `:!` (shell), 109
    `:t` (type), 108
comments, 107

defaulting, *see* type, defaulting
`drop`, 7, 8, 56, 69, 75

`elem`, 20, 34, 46, 47, 86, 92
endianness, 8, 9, 68
Enigma machine, 33, 99
    plugboard, 33
    reflector, 36
    rotor, 33

`False`, 1, 13
`fin`, *see* type, fin
float, 3
fold, 20, 21, 30, 35–38, 51, 80
function application, 17

Galois field, 50–52, 54, 59
`groupBy`, 7, 8, 15, 67, 69, 71

import directive, 25, 108
inference, *see* type, inference
integer, 2, 13

`join`, 7, 30, 31, 57, 67

known plaintext attack, 29

lambda expression, *see* λ-expression
`length`, 43, 85
`lg2`, 11, 16, 70
literate programming, 108

`max`, 11, 19, 73
`min`, 11, 12, 73
modular arithmetic, 11, 12, 41, 44, 71, 75, 76
module system, 25, 108
monomorphism, *see* type, monomorphism

overflow, 11, 75
overloading, *see* type, overloading

pattern matching, 35, 36
`pdiv`, *see* polynomial, division
plaintext, 27, 30
`pmod`, *see* polynomial, modulus
`pmult`, *see* polynomial, multiplication
polymorphism, *see* type, polymorphism
polynomial, 50
    addition, 50
    division, 51
    irreducable, 51
    modulus, 51
    multiplication, 51
    subtraction, 50
predicates, *see* type, predicates
private qualifier, 26, 108
`project`, 83
properties, 41
    `Q.E.D.`, 44
    `contradiction`, 44
    completeness, 46
    conditional, 45, 92
    counter-example, 44, 46
    function correspondence, 42
    polymorphic validity, 43
    proving, 44

rational, 3, 13
record, 10, 13, 37
recurrences, 18, 19
recursion, 18
`reverse`, 18, 37, 42, 57, 73, 84

satisfiability checking, 47
scytale, 30, 31, 81
sequence, 4, 13
    arithmetic lifting, 71
    cartesian, 5
    comprehensions, 5, 8, 19, 21, 36, 68, 71, 80
    enumerations, 5, 12, 28, 71
    finite, 6
    infinite, 6, 12, 13, 71
    nested, 5
    parallel, 5
settings
    `a`, (ASCII printing), 10, 28
    `base`, (output base), 2, 8, 63, 68
    `editor`, (file editor), 109
    `quickCheckCount`, 46, 87
    `sbv`, 88
shift cipher, 27
signature, *see* type, signature
`split`, 7, 8, 30, 31, 57, 67–69, 81
stream equations, 21
streams, 6, 19
string, 9
substitution cipher, 27, 29
    monoalphabetic, 29
    period, 39, 84
    polyalphabetic, 29, 33, 34, 39, 81
    polygraphic, 29
symmetric key, 49

`tail`, 14, 18, 36, 38, 73
`take`, 7, 8, 15, 68, 69
`transpose`, 7, 30, 56, 57, 67, 92
transposition cipher, 27, 30
`True`, 1, 10, 13, 74
tuple, 4, 10, 13
type
    `fin`, 15, 18, 74
    `inf`, 13, 15
    ambiguous, 81
    annotation, 45, 85
    defaulting, 2
    inference, 13, 107
    inline argument types, 25
    monomorphism, 13, 43
    overloading, 14
    polymorphism, 2, 14, 15, 107
    positional arguments, 24
    predicates, 15, 19, 29, 72, 74
    signature, 2, 7, 31, 41, 72, 73, 81, 85, 107, 108
    type classes, 22
    type context, 24
    type variables, 23
    undecidable, 16
type synonym, 21
    `Bool`, 22
    `String`, 22, 27
    `Word`, 22

undecidable, *see* type, undecidable
underflow, 11

Vigenère cipher, 28, 79

where clause, 17, 18, 28, 45, 86
while loop, 20
wildcard, *see* `_` (underscore)
word, 2, 8, 13
    arbitrary precision, 2

Yices, 44

`zero`, 10, 18, 42, 70, 73, 74, 85