Cryptol: A Comprehensive Guide

- Mastering the Art of Cryptol Programming -

Ji Yong-Hyeon

Department of Information Security, Cryptology, and Mathematics
College of Science and Technology
Kookmin University

March 19, 2024

Data

Table 1: Hierarchy of Physical and Logical Data Units

Physical Units	Logical Units
Bit (b)	Field (Set of bytes or words)
Nibble (4 bits)	Record (Collection of fields)
Byte (B)	Block (Groups of bytes or words for storage)
Half-word (2 Bytes)	File (Collection of records)
Word (4 Bytes)	Database (Organized collection of files)
Double-word (8 Bytes)	Data Warehouse (Aggregated collection of databases)

Cryptol vs EasyCrypt

- **Cryptol**: Cryptol is a domain-specific language designed specifically for specifying cryptographic algorithms. A creation by Galois, Inc., it's a tool used primarily for creating high-assurance cryptographic software. Cryptol allows developers to write cryptographic algorithms in a way that directly reflects the mathematical specifications, which makes it easier to analyze and verify for correctness and security.

- **EasyCrypt**: On the other hand, EasyCrypt is a toolset designed for the formal verification of cryptographic proofs. It provides a framework for developing and verifying mathematical proofs of the security of cryptographic constructions, such as encryption schemes, signature schemes, and hash functions. EasyCrypt operates at a higher level of abstraction compared to Cryptol and is used for proving the security properties of cryptographic protocols mathematically.

If you're comparing them from a user perspective, Cryptol is more about the implementation and specification of cryptographic algorithms, making sure they are implemented correctly according to their mathematical definitions. EasyCrypt is more about proving the theoretical security properties of cryptographic protocols and systems.

Contents

1 A Crash Course in Cryptol **4**
1.1 Basic Data Types 4
1.1.1 Bit: Booleans 4
1.1.2 Sequences: Homogeneous Collections 6
1.1.3 Tuples: Heterogeneous Collections 9
1.1.4 Records: Named Collections 12
1.1.5 Function 15
1.1.6 Type Variables 16

2 Functional Cryptography and Software Verification with Cryptol **18**
2.1 Introduction to Cryptol 18
2.2 Split and Join 21
2.3 Verification 22

Introduction

Overview

Cryptol is a domain-specific language that has been engineered with a clear purpose: to aid in the specification, implementation, and verification of cryptographic algorithms. Originating from a need to describe cryptographic functions in a manner that is both precise and verifiable, Cryptol has evolved as a bridge between the abstract world of cryptographic theory and the concrete realm of implementation.

Unlike general-purpose programming languages, which are designed to be versatile and broad, Cryptol is finely tuned for a specific domain. It embodies the principles of functional programming, making it an exemplary tool for cryptographers who are more accustomed to mathematical functions than to traditional software engineering constructs.

Functional Programming in Cryptol

Functional programming is a paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. Cryptol, inheriting this paradigm, allows developers to write code that closely resembles the mathematical definitions of cryptographic algorithms.

Pure Functions

At the heart of functional programming—and consequently at the core of Cryptol—are pure functions. These functions, which provide the same output for the same input every time, are devoid of side effects, ensuring predictability and ease of formal verification. In Cryptol, this principle allows for the straightforward expression of cryptographic transformations.

Immutability

Cryptol espouses the functional programming principle of immutability. Once data is created, it cannot be altered, mirroring the immutable nature of data in cryptographic processes. This property simplifies reasoning about the behavior of cryptographic algorithms and enhances the reliability and security of their implementations.

Type System and Type Inference

Cryptol's type system is robust and expressive, facilitating the prevention of common bugs found in cryptographic software. Types in Cryptol are not just annotations but are an integral part of the language's semantics, enforcing the correct usage of cryptographic constructs.

Furthermore, Cryptol's type inference mechanism reduces the verbosity of code, making algorithms concise and less prone to error.

Higher-order Functions and Currying

Cryptol supports higher-order functions, a hallmark of functional programming. These are functions that can take other functions as inputs or return them as outputs. This feature enables cryptographers to construct higher-level abstractions and compose algorithms elegantly. Additionally, Cryptol utilizes currying, allowing partial application of functions and further contributing to the language's expressiveness and flexibility.

Coding Style in Cryptol

The coding style in Cryptol is distinct and aligned with its functional nature. Developers are encouraged to write clear, concise, and mathematically-oriented code. The style promotes readability and maintainability, crucial for the complex and sensitive nature of cryptographic software.

Clarity and Conciseness

Cryptol code should be as clear and concise as possible, mirroring mathematical definitions. This clarity is achieved through the use of descriptive identifiers, minimal use of side effects, and leveraging Cryptol's powerful type system.

Modularization and Reuse

Cryptol encourages modularization and the reuse of code. Functions and types should be defined in a modular manner, allowing them to be easily reused across different cryptographic algorithms and protocols.

Testing and Verification

Given the critical nature of cryptographic software, testing and formal verification are integral parts of the Cryptol workflow. The language provides facilities for both property-based testing and formal verification, enabling developers to rigorously test and prove properties about their code.

Cryptol stands at the intersection of cryptography and functional programming. Its design philosophy, rooted in the principles of functional programming, makes it an ideal tool for specifying, implementing, and verifying cryptographic algorithms. By understanding and embracing the functional aspects and coding style of Cryptol, developers and cryptographers can create more secure, reliable, and verifiable cryptographic software.

Key Features

- **Mathematical Notation:** Cryptol's syntax closely mirrors the mathematical notation used in cryptography, making algorithms more readable and understandable.

- **Formal Verification:** Allows for the formal verification of algorithm properties, ensuring that they comply with their specifications.
- **Type System:** Features a strong, static type system that prevents many common errors in cryptographic implementations.
- **Abstraction:** Supports high levels of abstraction, enabling cryptographers to focus on algorithmic design rather than implementation details.

$$S = \left\{2n \mid n \in \mathbb{N}, 1 \leq n \leq 10\right\}$$

- $$\begin{array}{rccc} \mathsf{encrypt} & : & \{0,1\}^{128} & \longrightarrow & \{0,1\}^{128} \\ & & m & \longmapsto & m \oplus \delta \end{array}$$
- $$\begin{array}{rccc} \mathsf{decrypt} & : & \{0,1\}^{128} & \longrightarrow & \{0,1\}^{128} \\ & & c & \longmapsto & c \oplus \delta \end{array}$$

$$\mathsf{decrypt}(\mathsf{encrypt}(m)) = (m \oplus \delta) \oplus \delta = m$$

- Define a mapping

$$\mathsf{nagateByte} : \{0,1\}^8 \to \{0,1\}^8$$

where

$$\mathsf{negateByte}(b) := \neg b.$$

- Define a mapping

$$\mathsf{makeWord} : \{0,1\}^8 \times \{0,1\}^8 \times \{0,1\}^8 \times \{0,1\}^8 \to \{0,1\}^{32}$$

where

$$\mathsf{makeWord}(b_3, b_2, b_1, b_0) := b_3 \parallel b_2 \parallel b_1 \parallel b_0 \in \{0,1\}^{32}.$$

- Define a mapping

$$\mathsf{negateWord} : \{0,1\}^{32} \to \{0,1\}^{32}$$

where

$$\mathsf{negateWord}(w) := \neg w = \neg w_3 \parallel \neg w_2 \parallel \neg w_1 \parallel w_0.$$

Here, $w = w_3 \parallel w_2 \parallel w_1 \parallel w_0$

$$f : \{0,1\}^2 \times \{0,1\}^2 \to \{0,1\}^2.$$

Applications

Cryptol is widely used in academic and industrial settings for:

- Specifying cryptographic algorithms.
- Testing and verifying the correctness of cryptographic implementations.
- Educating new cryptographers about algorithm design and analysis.

Chapter 1

A Crash Course in Cryptol

1.1 Basic Data Types

1.1.1 Bit: Booleans

"The most basic data type in Cryptol, representing a single binary digit (0 or 1)."

Basic Bit Operations

Data Type: `Bit`

In Cryptol, a '`Bit`' can either be

'`True`' (equivalent to 1 in C) or '`False`' (equivalent to 0 in C).

Similar to C, Cryptol supports logical operations like **AND**, **OR**, and **NOT**.

```
$ Main> result_and
False
$ Main> result_or
True
$ Main> result_not
False
```

Intermediate Bit Operations

Conditional Expressions: Cryptol uses the '**if...then...else**' syntax, which is similar to the ternary operator '**?:**' in C.

XOR Operation: This is commonly used in cryptography for things like one-time pads.

```
a : Bit
a = True

b : Bit
b = False
```

```
// AND operation (similar to a & b in C)
result_and : Bit
result_and = a && b

// OR operation (similar to a | b in C)
result_or : Bit
result_or = a || b

// NOT operation (similar to !a in C)
result_not : Bit
result_not = ~a
```

```
// Conditional expression (similar to a ? b : c in C)
result_cond : Bit
result_cond = if a then b else ~b
```

Advanced Cryptographic Applications with Bit

Simple Cryptographic Operation: Here's a very basic example of a cryptographic operation in Cryptol, a one-time pad encryption of a single bit. In practical applications, you'll work with sequences or arrays of bits.

```
// XOR operation
c : Bit
c = True // Assuming some value

result_xor : Bit
result_xor = a ^ c // Similar to a ^ c in C
```

```
// One-time pad encryption
plaintext_bit : Bit
plaintext_bit = True // Your plaintext

key_bit : Bit
key_bit = False // Your secret key

// XOR for encryption
encrypted_bit : Bit
encrypted_bit = plaintext_bit ^ key_bit

// XOR for decryption (should equal plaintext_bit)
decrypted_bit : Bit
decrypted_bit = encrypted_bit ^ key_bit
```

```
module XOR where

// Defines an XOR function for two bits
xor : Bit -> Bit -> Bit
xor a b = (a || b) && ~(a && b)

// Property: XOR is its own inverse
property xor_inverse a b = xor (xor a b) b == a
```

```
$ cryptol XOR.cry
version 2.8.0

Loading module Cryptol
Loading module XOR
$ XOR> :prove xor_inverse
Q.E.D.
(Total Elapsed Time: 0.008s, using Z3)
```

```
seq : [4]Bit // A sequence of 4 bits
seq = [True, False, True, False] // Equivalent to 0b1010
```

```
element1 : Bit
element1 = seq @ 0 // Accessing the first element, similar to seq[0] in C

element3 : Bit
element3 = seq @ 3
```

1.1.2 Sequences: Homogeneous Collections

"Fixed-length, ordered collections of elements, which can be bits or other types."

Basic Sequence Operations

Sequence

In Cryptol, a **sequence** is defined with square brackets. `[, , ,]`.

Remark 1.1. Unlike C, where you typically define the array size and type, Cryptol infers these from the context.

```
$ Main> seq
0xa
```

Accessing Elements: Accessing sequence elements is similar to accessing array elements in C.

```
$ Main> element1
True
$ Main> element3
False
```

Intermediate Sequence Operations

Sequence Concatenation: You can concatenate sequences to form longer sequences.

```
$ Main> seq1
0x6
$ Main> seq2
```

```
seq1 : [3]Bit
seq1 = [True, True, False] // Similar to {1, 1, 0} in C

seq2 : [5]Bit
seq2 = [False, True, True, False, True] // Similar to {0, 1, 1, 0, 1} in C

// Concatenating two sequences 0b 110 01101 = 0b 1100 1101
combined_seq : [8]Bit
combined_seq = seq1 # seq2
```

```
seq1 : [3][4]
seq1 = [0xf, 0x1, 0x8]

seq2 : [5][8]
seq2 = [0xff, 0xcc, 0x11, 0x22, 0x99]

ele1 : [4]
ele1 = seq1 @ 1

ele2 : [8]
ele2 = seq2 @ 3
```

```
// Splitting a sequence into blocks
long_seq : [16]Bit // Similar to a block of data in cryptography
long_seq = 0xAAAA

blocks : [2][8]Bit // Splitting into two blocks of 8 bits
blocks = split long_seq // There's no direct C equivalent

joined_seq : [16]Bit
joined_seq = join blocks
```

```
0x0d
$ Main> combined_seq
0xcd
```

```
$ Main> seq1
[0xf, 0x1, 0x8]
$ Main> ele1
0x1
$Main> seq2
[0xff, 0xcc, 0x11, 0x22, 0x99]
$ Main> ele2
0x22
```

Advanced Cryptographic Applications with Sequence

Block Operations: In cryptography, you often work with blocks of data. In Cryptol, you can easily split sequences into blocks or combine blocks into a single sequence.

```
$ Main> long_seq
0xaaaa
$ Main> blocks
[0xaa, 0xaa]
$ Main> joined_seq
0xaaaa
```

Applying Functions to Sequences: In Cryptol, you can apply functions to entire sequences, which is useful for cryptographic transformations.

```
$ Main> negate_seq
<function>
```

```
// A simple bitwise NOT operation applied to a sequence
negate_seq : [8]Bit -> [8]Bit
negate_seq input = ~input // Bitwise NOT each element

input_seq : [8]Bit
input_seq = 0xCD // ~ 0b11001101 = 0b00110010 = 0x32

negated_seq : [8]Bit
negated_seq = negate_seq input_seq
```

```
$ Main> input_seq
0xcd
$ Main> negated_seq
0x32
```

```
// Example of XORing two sequences (common in many cryptographic algorithms)
xor_sequences : [8]Bit -> [8]Bit -> [8]Bit
xor_sequences seq1 seq2 = zipWith (^) seq1 seq2 // Element-wise XOR

seqA : [8]Bit
seqA = 0xAA

seqB : [8]Bit
seqB = 0x55

//   0b 1010 1010
// ^ 0b 0101 0101
//   0b 1111 1111
result_seq : [8]Bit
result_seq = xor_sequences seqA seqB // Result of XORing seqA and seqB
```

```
// A tuple containing a Bit, a 3-Bit sequence, and a 4-Bit sequence
myTuple : (Bit, [3]Bit, [4]Bit)
myTuple = (True, [False, True, False], [True, True, False, False])
```

Advanced Cryptographic Patterns: Sequences are used in various cryptographic algorithms, such as block ciphers and hash functions, to handle data in structured formats.

```
$ Main> seqA
0xaa
$ Main> seqB
0x55
$ Main> result_seq
0xff
```

Note (‘zipWith’ in Haskell).

```
1 // [f(x1, y1), f(x2, y2), ..., f(x_n, y_n)]
2 zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
3 zipWith f list1 list2
```

1.1.3 Tuples: Heterogeneous Collections

“Collections of possibly different types grouped together.”

Basic Tuple Operations

Tuple

A **tuple** can hold a fixed number of elements, which can be of different types.

Remark 1.2. Unlike C, where you define a struct with named fields, Cryptol tuples are indexed by position.

```
$ Main> myTuple
(True, 0x2, 0xc)
```

```
first_element : Bit
first_element = myTuple.0 // Accessing the first element of the tuple

second_element : [3]Bit
second_element = myTuple.1 // Accessing the second element of the tuple
```

```
// Updating the second element
updatedTuple : (Bit, [3]Bit, [4]Bit)
updatedTuple = (myTuple.0, [True, True, True], myTuple.2)
```

Accessing Tuple Elements: Elements in a tuple are accessed by their positions, starting from zero.

```
$ Main> first_element
True
$ Main> second_element
0x2
```

Updating Tuple Elements: Unlike in C, where you might update an element directly, in Cryptol, you usually create a new tuple with the updated value due to its immutable nature.

```
$ Main> updatedTuple
(True, 0x7, 0xc)
```

Intermediate Tuple Operations

Tuple Decomposition: You can decompose a tuple into its components, similar to unpacking a struct in C.

```
$ Main> bitVal
True
$ Main> threeBitSeq
0x2
$ Main> fourBitSeq
0xc
```

Nested Tuples: Tuples can contain other tuples, providing a way to structure complex data.

```
$ Main> nestedTuple
((True, False), 0xc)
```

```
// Decomposing the tuple into variables
(bitVal, threeBitSeq, fourBitSeq) = myTuple
```

```
// A tuple where the first element is also a tuple
nestedTuple : ((Bit, Bit), [4]Bit)
nestedTuple = ((True, False), [True, True, False, False])
```

```
// An example with a simple encryption scheme
type Key = [8]Bit
type State = [8]Bit
type CipherText = [8]Bit

encrypt : (Key, State) -> CipherText
encrypt (key, state) = zipWith (^) key state // A simple XOR-based encryption

property EncryptDecrypt =
  \(key : Key) (state : State) ->
    (encrypt(key, encrypt(key, state))) == state
```

Advanced Cryptographic Applications with Tuple

Using Tuples in Cryptographic Algorithms: Tuples can represent structured data in cryptographic algorithms, like keys and states.

```
$ Main> :prove
:prove EncryptDecrypt
        Q.E.D.
(Total Elapsed Time: 0.012s, using Z3)
```

Advanced Data Structuring: In more complex cryptographic algorithms, you might use tuples to represent different parts of the algorithm's state, input, or output.

```
$ Main> :prove
:prove DoubleUpdate
        Q.E.D.
(Total Elapsed Time: 0.013s, using Z3)
```

```
// An example using tuples for algorithm state management
type RoundKey = [8]Bit;
type RoundState = ([8]Bit, [8]Bit); // Tuple representing state for two rounds

// Update both states
updateState : (RoundKey, RoundState) -> RoundState;
updateState (key, (state1, state2)) =
  (zipWith (^) key state1, zipWith (^) key state2);

property DoubleUpdate =
  \(key : RoundKey) (initialState : RoundState) ->
    updateState(key, updateState(key, initialState)) == initialState
```

```
type MyRecord = {
  field1 : Bit,
  field2 : [3]Bit,
  field3 : [4]Bit
}

myRecord : MyRecord
myRecord = {
  field1 = True,
  field2 = [False, True, False],
  field3 = [True, True, False, False]
}

first_field : Bit
first_field = myRecord.field1 // Accessing the first field of the record

second_field : [3]Bit
second_field = myRecord.field2 // Accessing the second field of the record
```

```
type NestedRecord = {nested : MyRecord, anotherField : [5]Bit}
nestedRecord : NestedRecord
nestedRecord = {
  nested = {
    field1 = False,
    field2 = [True, False, True],
    field3 = [False, False, False, True]
  },
  anotherField = [True, True, True, False, False]
}
```

1.1.4 Records: Named Collections

"Similar to tuples, but each element is identified by a name."

Basic Record Operations

Record

In Cryptol, you define a record with braces '{}' and specify the names and types of its fields.

```
$ Main> myRecord
{field1 = True, field2 = 0x2, field3 = 0xc}
$ Main> first_field
True
$ Main> second_field
0x2
```

Intermediate Record Operations

Nested Records: Records can contain other records, which allows you to structure complex data hierarchically.

```
$ Main> nestedRecord
{nested = {field1 = False, field2 = 0x5, field3 = 0x1},
        anotherField = 0x1c}
```

```
// A record type for encryption configuration
type CryptoConfig = {key : [128], iv : [128]}

// An example configuration using all zeros (False)
defaultConfig : CryptoConfig
defaultConfig = {key = zero, iv = ~zero}
```

```
type AlgorithmState = {counter : [32], buffer : [64], flags : [8]} // A record
    type for the algorithm state

initialState : AlgorithmState
initialState = {counter = zero, buffer = zero, flags = zero} // Initializes all to
     zero

updateState : AlgorithmState -> AlgorithmState
updateState state =  { state |
  counter = state.counter + 1,
  buffer = ~state.buffer,
  flags = zipWith (^) state.flags 0xf0
}
```

Advanced Cryptographic Applications with Record

Cryptographic Keys and Configurations: Records are excellent for representing complex keys and configuration options in cryptographic algorithms.

```
$ Main> defaultConfig
{key = 0x00000000000000000000000000000000,
 iv = 0xffffffffffffffffffffffffffffffff}
```

State Management in Cryptographic Algorithms: Use records to manage the state of an algorithm, making it easier to handle multiple pieces of related data.

```
$ Main> :prove
:prove CounterIncrement
Q.E.D.
(Total Elapsed Time: 0.013s, using Z3)
:prove BufferToggle
Q.E.D.
(Total Elapsed Time: 0.008s, using Z3)
:prove FlagsXOR
Q.E.D.
(Total Elapsed Time: 0.009s, using Z3)
```

```
property CounterIncrement state =
  (updateState state).counter == state.counter + 1

property BufferToggle state =
  (updateState (updateState state)).buffer == state.buffer

property FlagsXOR state =
  (updateState state).flags == zipWith (^) state.flags 0xf0
```

```
add : Integer -> Integer -> Integer;
add x y = x + y;
```

```
andBits : [8] -> [8] -> [8];
andBits a b = zipWith (&&) a b;
```

1.1.5 Function

"Types representing functions from inputs to outputs."

Basic Function Usage

Simple Function: Let's start with a basic function that adds two numbers. In Cryptol, the syntax for function definition is concise.

```
$ Main> add 4294967296 4294967296
8589934592
```

Function with Bitwise Operation: Functions in Cryptol commonly operate on bits and sequences of bits. Here's a function that performs a bitwise AND on two 8-bit sequences.

```
$ Main> andBits 0x0f 0xf0
0x00
```

Intermediate Function Usage

Functions with Conditional Logic: Cryptol functions can implement more complex logic, including conditionals.

```
$ Main> max 51 94
94
```

Recursive Functions: Cryptol supports recursive functions, which can be powerful for certain types of algorithms.

```
Main> factorial 10
3628800
```

```
max : Integer -> Integer -> Integer
max x y = if x > y then x else y
```

```
property max_ge_x (x : Integer, y : Integer) = max x y >= x

property max_ge_y (x : Integer, y : Integer) = max x y >= y

property max_is_either (x : Integer, y : Integer) =
  (max x y == x) || (max x y == y)
```

```
factorial : Integer -> Integer;
factorial n = if n == 0 then 1 else n * factorial (n - 1);
```

Advanced Cryptographic Function Usage

Encrypting with XOR (One-Time Pad): Here's how you might define a simple encryption function using XOR, which is a foundational operation in many cryptographic systems..

```
$ Main> :prove
:prove Reversibility
        Q.E.D.
(Total Elapsed Time: 0.012s, using Z3)
```

1.1.6 Type Variables

Basic Usage of Type Variables

Generic Identity Function: A simple example of a type variable in use is the identity function, which returns whatever value is passed to it, regardless of type.

```
$ Main> id 0xff
0xff
$ Main> id 0xffff
0xffff
```

Generic Swap Function: This function swaps the elements of a 2-tuple, and works for any types contained in the tuple.

```
$ Main> mixedPair
(42, 0xaa)
$ Main> swappedMixedPairnewp
(0xaa, 42)
```

```
xorEncrypt : [8] -> [8] -> [8];
xorEncrypt key plaintext = zipWith (^) plaintext key;

property Reversibility (key : [8], plaintext : [8]) =
  xorEncrypt key (xorEncrypt key plaintext) == plaintext
```

```
id : {a} a -> a;
id x = x;
```

```
swap : {a, b} (a, b) -> (b, a)
swap (x, y) = (y, x)

mixedPair : (Integer, [8])
mixedPair = (42, 0xaa)

swappedMixedPair : ([8], Integer)
swappedMixedPair = swap mixedPair
```

Intermediate Usage of Type Variables

Generic Arrays: You can define functions that operate on arrays of any type. This is useful for functions that perform operations like reversing an array.

```
$ Main> id 0xff
0xff
$ Main> id 0xffff
0xffff
```

```
id : {a} a -> a
id x = x
```

Chapter 2

Functional Cryptography and Software Verification with Cryptol

2.1 Introduction to Cryptol

Cryptol is a purely functional programming language for cryptographic specifications. It's got a large amount of Haskell influence.

- Very syntactically similar to Haskell
- And a similar type system (with some additions, and reductions)
- Purely functional
- Clearly the second best programming language by these metrics alone.

Some additions.

- Type level arithmetic, and a solver for it
- Verification technology
- A smattering of other small scoping

Some reductions.

- It has type classes, but you can't write them.
- No built in IO or any other operations.
- No user-defined data type or "pattern matching" beyond the built-in sequence type.

C:

```c
uint32_t f(uint32_t x, uint32_t y) {
    return x + y;
}
```

Haskell:

```haskell
f :: Word32 -> Word32 -> Word32
f x y = x + y
```

Cryptol:

```
f : [32] -> [32] -> [32]
f x y = x + y
```

Large or “weird” numbers.

```
f : [17] -> [17]
f x = -x

g : [23424] -> [548683776]
g x = x * x
```

Sequence are fundamental

```
[0 .. 3] = [0, 1, 2, 3]
f : { x } x -> [4]x
f x = [x, x, x, x]
// Define a sequence of 5 integers
sequenceOfInts : [5]Integer
sequenceOfInts = [1, 2, 3, 4, 5]

// Define a sequence of bits (binary sequence)
sequenceOfBits : [8]Bit
sequenceOfBits = [1, 0, 1, 1, 0, 0, 1, 0]

// Define a nested sequence (matrix of bits)
matrixOfBits : [3][4]Bit
matrixOfBits = [[1, 0, 1, 0], [0, 1, 0, 1], [1, 1, 1, 1]]

// Accessing elements and slices
firstElement = sequenceOfInts @ 0      // Access the first element (index starts
    from 0)
firstTwoBits = sequenceOfBits @ [0..1] // Access the first two bits
secondRow = matrixOfBits @ 1           // Access the second row of the matrix

// Sequence operations
reversedSequence = reverse sequenceOfInts    // Reverse the sequence
concatenatedSequence = sequenceOfInts # [6] // Concatenate [6] to the end of the
    sequence
```

```
// Define a sequence of 16 bits
myBits : [16]Bit
myBits = [1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1]

// Split the sequence into chunks of 4 bits
splitChunks : [4][4]Bit
splitChunks = split(myBits)

// Define a list of sequences
myChunks : [4][4]Bit
myChunks = [[1, 0, 1, 1], [0, 0, 1, 0], [1, 1, 1, 0], [1, 0, 0, 1]]

// Join the chunks into a single sequence
joinedSequence : [16]Bit
joinedSequence = join(myChunks)
```

2.2 Split and Join

```c
// Pseudo-C code representing what 'g' might look like in C
void g(type *z, type (*result)[a], int a) {
    // Assuming 'z' is an array of length 2*a and 'type' is whatever 'b' represents
    // 'result' is an array of two elements, each of which can hold 'a' items of 'type'
    for (int i = 0; i < a; i++) {
        result[0][i] = z[i];      // Copy the first half of 'z' into 'result[0]'
        result[1][i] = z[i + a];  // Copy the second half of 'z' into 'result[1]'
    }
}
```

In this pseudo-C version, g takes an array z and splits it into two halves, storing these in result[0] and result[1]. This is analogous to the Cryptol version where split takes z and returns two sequences, y and x, which get assigned in reverse order due to the pattern y, x in the Cryptol code.

```
// Split into four bytes (8-bit)
split 0xAABBCCDD : [4][8] = 0xAA, 0xBB, 0xCC, 0xDD
// Concatenation
join [0xAA, 0xBB, 0xCC, 0xDD] = 0xAABBCCDD
```

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

g: (a, b) (fin a) => [2 * a]b -> [2][a]b
g z = {x, y}
  where {y, x} = split z
```

2.3 Verification

The Quarter Round - C99

```
static void
qround(uint32_t* x, // Pointer to an array
       uint32_t a, // Index of 'x', representing 'a'
       uint32_t b, // Second index
       uint32_t c, // ...
       uint32_t d, // ...
) {
    /* L32 is left 32-bit ROLL, not a shift */
    x[a] += x[b]; x[d] = L32(x[d] ^ x[a], 16);
    x[c] += x[d]; x[b] = L32(x[b] ^ x[c], 12);
    x[a] += x[b]; x[d] = L32(x[d] ^ x[a],  0);
    x[c] += x[d]; x[b] = L32(x[b] ^ x[c],  7);
}
```

Cryptol has a sister tool, called **SAW**, which can do this using deep black magic that we dare not speak of (yet).

SAW has its own typed language - called SAWScript - used for scripting proofs in an automated way, which we'll use.

SAW doesn't work on surface-level syntax. It works with intermediate representations, often ones your compiler splits out. In particular, SAW comes with tools for ingesting **LLVM bitcode, JVM Bytecode and Cryptol programs**.

Bibliography

[1] "Cryptography and Verification with Cryptol" YouTube, uploaded by Compose Conference, 03 May 2016, https://www.youtube.com/watch?v=sC2_5WaavFc