Cryptol in *n* Minutes

Frank Seaton Taylor

July 24, 2013

Cryptol

- is a domain specific language and tool suite.

Cryptol

- is a domain specific language and tool suite.
- was created by Galois, Inc with support from NSA cryptographers.

Cryptol

- is a domain specific language and tool suite.
- was created by Galois, Inc with support from NSA cryptographers.
- has lots of cool programming language features:

Cryptol

- is a domain specific language and tool suite.
- was created by Galois, Inc with support from NSA cryptographers.
- has lots of cool programming language features:
  - strong, static typing

Cryptol

- is a domain specific language and tool suite.
- was created by Galois, Inc with support from NSA cryptographers.
- has lots of cool programming language features:
  - strong, static typing
  - type inference

Cryptol

- is a domain specific language and tool suite.
- was created by Galois, Inc with support from NSA cryptographers.
- has lots of cool programming language features:
  - strong, static typing
  - type inference
  - parametric size-polymorphism

Cryptol

- is a domain specific language and tool suite.
- was created by Galois, Inc with support from NSA cryptographers.
- has lots of cool programming language features:
  - strong, static typing
  - type inference
  - parametric size-polymorphism
  - higher-order functions

Cryptol

- is a domain specific language and tool suite.
- was created by Galois, Inc with support from NSA cryptographers.
- has lots of cool programming language features:
  - strong, static typing
  - type inference
  - parametric size-polymorphism
  - higher-order functions
- is used for gold standard algorithm specs, evaluations against those and exploration.

Cryptol

- is a domain specific language and tool suite.
- was created by Galois, Inc with support from NSA cryptographers.
- has lots of cool programming language features:
  - strong, static typing
  - type inference
  - parametric size-polymorphism
  - higher-order functions
- is used for gold standard algorithm specs, evaluations against those and exploration.
- is available via: `http://www.cryptol.net`

Data and Programs

- Everything is a n-dimensional structure of bits.

Data and Programs

- Everything is a n-dimensional structure of bits[1].

[1] Most everything, anyway.

Data and Programs

- Everything is a n-dimensional structure of bits[1].
  - 0-d `False : Bit` `True : Bit`

[1] Most everything, anyway.

Data and Programs

- Everything is a n-dimensional structure of bits[1].
  - 0-d `False : Bit` `True : Bit`
  - 1-d[2] `0xaa : [8]` `0b1011 : [4]`

[1] Most everything, anyway.

Data and Programs

- Everything is a *n*-dimensional structure of bits[1].
  - 0-d `False : Bit` `True : Bit`
  - 1-d[2] `0xaa : [8]` `0b1011 : [4]`

[1] Most everything, anyway.

[2] 1-d sequences of bits are nonnegative integers ("words") in arithmetic contexts.

Data and Programs

- Everything is a n-dimensional structure of bits[1].
  - 0-d `False : Bit` `True : Bit`
  - 1-d[2] `0xaa : [8]` `0b1011 : [4]`
  - 2-d `[42 0b01010101 0xaa 0o377] : [4][8]`

[1] Most everything, anyway.

[2] 1-d sequences of bits are nonnegative integers ("words") in arithmetic contexts.

Data and Programs

- Everything is a n-dimensional structure of bits[1].
  - 0-d `False : Bit` `True : Bit`
  - 1-d[2] `0xaa : [8]` `0b1011 : [4]`
  - 2-d `[42 0b01010101 0xaa 0o377] : [4][8]`
  - 3-d `[[0 1] [1 2] [3 5] [8 13]] : [4][2][4]`

[1] Most everything, anyway.

[2] 1-d sequences of bits are nonnegative integers ("words") in arithmetic contexts.

Data and Programs

- Everything is a n-dimensional structure of bits[1].
  - 0-d `False : Bit` `True : Bit`
  - 1-d[2] `0xaa : [8]` `0b1011 : [4]`
  - 2-d `[42 0b01010101 0xaa 0o377] : [4][8]`
  - 3-d `[[0 1] [1 2] [3 5] [8 13]] : [4][2][4]`
  - ...

[1] Most everything, anyway.

[2] 1-d sequences of bits are nonnegative integers ("words") in arithmetic contexts.

Data and Programs

- Everything is a n-dimensional structure of bits[1].
  - 0-d `False : Bit` `True : Bit`
  - 1-d[2] `0xaa : [8]` `0b1011 : [4]`
  - 2-d `[42 0b01010101 0xaa 0o377] : [4][8]`
  - 3-d `[[0 1] [1 2] [3 5] [8 13]] : [4][2][4]`
  - ...
- Programs are a sequence of mathematical definitions.

[1] Most everything, anyway.

[2] 1-d sequences of bits are nonnegative integers ("words") in arithmetic contexts.

Data and Programs

- Everything is a *n*-dimensional structure of bits[1].
  - 0-d `False : Bit` `True : Bit`
  - 1-d[2] `0xaa : [8]` `0b1011 : [4]`
  - 2-d `[42 0b01010101 0xaa 0o377] : [4][8]`
  - 3-d `[[0 1] [1 2] [3 5] [8 13]] : [4][2][4]`
  - ...
- Programs are a sequence of mathematical definitions.
  - Definitions may be accompanied by a type signature (if not, one is *usually* inferred).

[1] Most everything, anyway.

[2] 1-d sequences of bits are nonnegative integers ("words") in arithmetic contexts.

Data and Programs

- Everything is a *n*-dimensional structure of bits[1].
  - 0-d `False : Bit` `True : Bit`
  - 1-d[2] `0xaa : [8]` `0b1011 : [4]`
  - 2-d `[42 0b01010101 0xaa 0o377] : [4][8]`
  - 3-d `[[0 1] [1 2] [3 5] [8 13]] : [4][2][4]`
  - ...
- Programs are a sequence of mathematical definitions.
  - Definitions may be accompanied by a type signature (if not, one is *usually* inferred).
  - Everything must have a size that is known or inferred at "compile time".

[1] Most everything, anyway.

[2] 1-d sequences of bits are nonnegative integers ("words") in arithmetic contexts.

Data and Programs

- Everything is a *n*-dimensional structure of bits[1].
  - 0-d `False : Bit` `True : Bit`
  - 1-d[2] `0xaa : [8]` `0b1011 : [4]`
  - 2-d `[42 0b01010101 0xaa 0o377] : [4][8]`
  - 3-d `[[0 1] [1 2] [3 5] [8 13]] : [4][2][4]`
  - ...
- Programs are a sequence of mathematical definitions.
  - Definitions may be accompanied by a type signature (if not, one is *usually* inferred).
  - Everything must have a size that is known or inferred at "compile time".
  - Definitions are computationally neutral. Cryptol tools provide the computational context (interpreters, compilers, code generators, SAT solvers, theorem provers, etc.).

[1] Most everything, anyway.

[2] 1-d sequences of bits are nonnegative integers ("words") in arithmetic contexts.

Operators[3]

- of type `a -> a`

[3] Types on this slide are simplified for illustrative purposes.

Operators[3]

- of type `a -> a`
  - `~` (bitwise inversion)

[3] Types on this slide are simplified for illustrative purposes.

Operators[3]

- of type `a -> a`
  - `~` (bitwise inversion)
- of type `(a,a) -> a`

[3] Types on this slide are simplified for illustrative purposes.

Operators[3]

- of type `a -> a`
  - `~` (bitwise inversion)
- of type `(a,a) -> a`
  - `&` `|` `^` (bitwise logical operations)

[3] Types on this slide are simplified for illustrative purposes.

Operators[3]

- of type `a -> a`
  - `~` (bitwise inversion)
- of type `(a,a) -> a`
  - `&` `|` `^` (bitwise logical operations)
- of type `(a,a) -> Bit`

[3] Types on this slide are simplified for illustrative purposes.

Operators[3]

- of type `a -> a`
  - `~` (bitwise inversion)
- of type `(a,a) -> a`
  - `&` `|` `^` (bitwise logical operations)
- of type `(a,a) -> Bit`
  - `==` `!=` (structural comparison)

[3] Types on this slide are simplified for illustrative purposes.

Operators[3]

- of type `a -> a`
  - `~` (bitwise inversion)
- of type `(a,a) -> a`
  - `&` `|` `^` (bitwise logical operations)
- of type `(a,a) -> Bit`
  - `==` `!=` (structural comparison)
- of type `([a],[a]) -> Bit`

[3] T

Operators[3]

- of type `a -> a`
  - `~` (bitwise inversion)
- of type `(a,a) -> a`
  - `&` `|` `^` (bitwise logical operations)
- of type `(a,a) -> Bit`
  - `==` `!=` (structural comparison)
- of type `([a],[a]) -> Bit`
  - `>=` `>` `<=` `<` (nonnegative word comparisons)

[3] ...

Operators[3]

- of type `a -> a`
  - `~` (bitwise inversion)
- of type `(a,a) -> a`
  - `&` `|` `^` (bitwise logical operations)
- of type `(a,a) -> Bit`
  - `==` `!=` (structural comparison)
- of type `([a],[a]) -> Bit`
  - `>=` `>` `<=` `<` (nonnegative word comparisons)
- of type `([a]b,[a]b) -> [a]b`

Operators[3]

- of type `a -> a`
  - `~` (bitwise inversion)
- of type `(a,a) -> a`
  - `&` `|` `^` (bitwise logical operations)
- of type `(a,a) -> Bit`
  - `==` `!=` (structural comparison)
- of type `([a],[a]) -> Bit`
  - `>=` `>` `<=` `<` (nonnegative word comparisons)
- of type `([a]b,[a]b) -> [a]b`
  - `+` `-` `*` `/` `%` `**` (wordwise modular arithmetic)

Operators[3]

- of type `a -> a`
  - `~` (bitwise inversion)
- of type `(a,a) -> a`
  - `&` `|` `^` (bitwise logical operations)
- of type `(a,a) -> Bit`
  - `==` `!=` (structural comparison)
- of type `([a],[a]) -> Bit`
  - `>=` `>` `<=` `<` (nonnegative word comparisons)
- of type `([a]b,[a]b) -> [a]b`
  - `+` `-` `*` `/` `%` `**` (wordwise modular arithmetic)
- of type `([a]b,[c]) -> [a]b`

Operators[3]

- of type `a -> a`
  - `~` (bitwise inversion)
- of type `(a,a) -> a`
  - `&` `|` `^` (bitwise logical operations)
- of type `(a,a) -> Bit`
  - `==` `!=` (structural comparison)
- of type `([a],[a]) -> Bit`
  - `>=` `>` `<=` `<` (nonnegative word comparisons)
- of type `([a]b,[a]b) -> [a]b`
  - `+` `-` `*` `/` `%` `**` (wordwise modular arithmetic)
- of type `([a]b,[c]) -> [a]b`
  - `>>` `>>>` `<<<` `<<` (shifts and rotates)

Operators[3]

- of type `a -> a`
  - `~` (bitwise inversion)
- of type `(a,a) -> a`
  - `&` `|` `^` (bitwise logical operations)
- of type `(a,a) -> Bit`
  - `==` `!=` (structural comparison)
- of type `([a],[a]) -> Bit`
  - `>=` `>` `<=` `<` (nonnegative word comparisons)
- of type `([a]b,[a]b) -> [a]b`
  - `+` `-` `*` `/` `%` `**` (wordwise modular arithmetic)
- of type `([a]b,[c]) -> [a]b`
  - `>>` `>>>` `<<<` `<<` (shifts and rotates)
- of type `([a]b,[c]b) -> [a+c]b`

Operators[3]

- of type `a -> a`
  - `~` (bitwise inversion)
- of type `(a,a) -> a`
  - `&` `|` `^` (bitwise logical operations)
- of type `(a,a) -> Bit`
  - `==` `!=` (structural comparison)
- of type `([a],[a]) -> Bit`
  - `>=` `>` `<=` `<` (nonnegative word comparisons)
- of type `([a]b,[a]b) -> [a]b`
  - `+` `-` `*` `/` `%` `**` (wordwise modular arithmetic)
- of type `([a]b,[c]) -> [a]b`
  - `>>` `>>>` `<<<` `<<` (shifts and rotates)
- of type `([a]b,[c]b) -> [a+c]b`
  - `#` (concatenation)

Operators[3]

- `~` (bitwise inversion)
- `&` `|` `^` (bitwise logical operations)
- `==` `!=` (structural comparison)
- `>=` `>` `<=` `<` (nonnegative word comparisons)
- `+` `-` `*` `/` `%` `**` (wordwise modular arithmetic)
- `>>` `>>>` `<<<` `<<` (shifts and rotates)
- `#` (concatenation)

[3] Types on this slide are simplified for illustrative purposes.

Operators[3]

- `~` (bitwise inversion)
- `&` `|` `^` (bitwise logical operations)
- `==` `!=` (structural comparison)
- `>=` `>` `<=` `<` (nonnegative word comparisons)
- `+` `-` `*` `/` `%` `**` (wordwise modular arithmetic)
- `>>` `>>>` `<<<` `<<` (shifts and rotates)
- `#` (concatenation)

- of type `([a]b,[c]) -> b`

[3] Types on this slide are simplified for illustrative purposes.

Operators[3]

- `~` (bitwise inversion)
- `&` `|` `^` (bitwise logical operations)
- `==` `!=` (structural comparison)
- `>=` `>` `<=` `<` (nonnegative word comparisons)
- `+` `-` `*` `/` `%` `**` (wordwise modular arithmetic)
- `>>` `>>>` `<<<` `<<` (shifts and rotates)
- `#` (concatenation)

- of type `([a]b,[c]) -> b`
  - `@` `!` (0-based indexing from front or rear)

[3] Types on this slide are simplified for illustrative purposes.

Operators[3]

- `~` (bitwise inversion)
- `&` `|` `^` (bitwise logical operations)
- `==` `!=` (structural comparison)
- `>=` `>` `<=` `<` (nonnegative word comparisons)
- `+` `-` `*` `/` `%` `**` (wordwise modular arithmetic)
- `>>` `>>>` `<<<` `<<` (shifts and rotates)
- `#` (concatenation)
- `@` `!` (0-based indexing from front or rear)

- of type `([a]b,[c][d]) -> [c]b`

[3] Types on this slide are simplified for illustrative purposes.

Operators[3]

- `~` (bitwise inversion)
- `&` `|` `^` (bitwise logical operations)
- `==` `!=` (structural comparison)
- `>=` `>` `<=` `<` (nonnegative word comparisons)
- `+` `-` `*` `/` `%` `**` (wordwise modular arithmetic)
- `>>` `>>>` `<<<` `<<` (shifts and rotates)
- `#` (concatenation)
- `@` `!` (0-based indexing from front or rear)

- of type `([a]b,[c][d]) -> [c]b`
  - `@@` `!!` (0-based slicing from front or rear)

[3] Types on this slide are simplified for illustrative purposes.

Operators

- `~` (bitwise inversion)
- `&` `|` `^` (bitwise logical operations)
- `==` `!=` (structural comparison)
- `>=` `>` `<=` `<` (nonnegative word comparisons)
- `+` `-` `*` `/` `%` `**` (wordwise modular arithmetic)
- `>>` `>>>` `<<<` `<<` (shifts and rotates)
- `#` (concatenation)
- `@` `!` (0-based indexing from front or rear)
- `@@` `!!` (0-based slicing from front or rear)

Primitives

- `take` `drop` (get the front or back of a list)

Primitives

- `take` `drop` (get the front or back of a list)
- `tail` (drops the first item of a list)

Primitives

- `take` `drop` (get the front or back of a list)
- `tail` (drops the first item of a list)
- `split` `join` (e.g. word to octets or vice-versa)

Primitives

- `take` `drop` (get the front or back of a list)
- `tail` (drops the first item of a list)
- `split` `join` (e.g. word to octets or vice-versa)
- `splitBy` `groupBy` (variations of `split`)

Primitives

- `take` `drop` (get the front or back of a list)
- `tail` (drops the first item of a list)
- `split` `join` (e.g. word to octets or vice-versa)
- `splitBy` `groupBy` (variations of `split`)
- `reverse` (sequence operation)

Primitives

- `take` `drop` (get the front or back of a list)
- `tail` (drops the first item of a list)
- `split` `join` (e.g. word to octets or vice-versa)
- `splitBy` `groupBy` (variations of `split`)
- `reverse` (sequence operation)
- `transpose` (array operation)

Primitives

- `take` `drop` (get the front or back of a list)
- `tail` (drops the first item of a list)
- `split` `join` (e.g. word to octets or vice-versa)
- `splitBy` `groupBy` (variations of `split`)
- `reverse` (sequence operation)
- `transpose` (array operation)
- `negate` (two's complement)

Primitives

- `take` `drop` (get the front or back of a list)
- `tail` (drops the first item of a list)
- `split` `join` (e.g. word to octets or vice-versa)
- `splitBy` `groupBy` (variations of `split`)
- `reverse` (sequence operation)
- `transpose` (array operation)
- `negate` (two's complement)
- `zero` (structure of `False` bits of any type)

Primitives

- `take` `drop` (get the front or back of a list)
- `tail` (drops the first item of a list)
- `split` `join` (e.g. word to octets or vice-versa)
- `splitBy` `groupBy` (variations of `split`)
- `reverse` (sequence operation)
- `transpose` (array operation)
- `negate` (two's complement)
- `zero` (structure of `False` bits of any type)
- `min` `max` `parity` (obvious)

Primitives

- `take` `drop` (get the front or back of a list)
- `tail` (drops the first item of a list)
- `split` `join` (e.g. word to octets or vice-versa)
- `splitBy` `groupBy` (variations of `split`)
- `reverse` (sequence operation)
- `transpose` (array operation)
- `negate` (two's complement)
- `zero` (structure of `False` bits of any type)
- `min` `max` `parity` (obvious)
- `lg2` `width` (esoteric)

Functions

```
min1 : ([16],[16]) -> [16]; // type signature
min1 (x, y) = if 1 >= x     // definition
  then x
  else min (x, y);
```

Functions

```
min1 : ([16],[16]) -> [16]; // type signature
min1 (x, y) = if 1 >= x     // definition
  then x
  else min (x, y);
```

Using Cryptol's interpreter, we see:

`min1(2, 1/0)` $\rightsquigarrow$ ERROR

Functions

```
min1 : ([16],[16]) -> [16]; // type signature
min1 (x, y) = if 1 >= x     // definition
  then x
  else min (x, y);
```

Using Cryptol's interpreter, we see:

`min1(2, 1/0)` $\rightsquigarrow$ ERROR

and

`min1(0, 1/0)` $\rightsquigarrow$

Functions

```
min1 : ([16],[16]) -> [16]; // type signature
min1 (x, y) = if 1 >= x     // definition
  then x
  else min (x, y);
```

Using Cryptol's interpreter, we see:

`min1(2, 1/0)` $\rightsquigarrow$ ERROR

and

`min1(0, 1/0)` $\rightsquigarrow$ 0

Functions and Laziness

```
min1 : ([16],[16]) -> [16]; // type signature
min1 (x, y) = if 1 >= x     // definition
  then x
  else min (x, y);
```

Using Cryptol's interpreter, we see:

`min1(2, 1/0)` $\rightsquigarrow$ ERROR

and

`min1(0, 1/0)` $\rightsquigarrow$ 0

- Cryptol is lazy. (Values are computed on demand.)

Functions and Laziness

```
min1 : ([16],[16]) -> [16]; // type signature
min1 (x, y) = if 1 >= x     // definition
  then x
  else min (x, y);
```

Using Cryptol's interpreter, we see:

`min1(2, 1/0)` $\rightsquigarrow$ ERROR

and

`min1(0, 1/0)` $\rightsquigarrow$ 0

- Cryptol is lazy. (Values are computed on demand.)
- Cryptol supports conceptually infinite lists. (So long as only a finite prefix is demanded everything is fine.)

Enumerations

- `[0 .. 5]` $\rightsquigarrow$ `[0 1 2 3 4 5]`

Enumerations

- `[0 .. 5]` $\rightsquigarrow$ `[0 1 2 3 4 5]`
- `[0 2 .. 5]` $\rightsquigarrow$ `[0 2 4]`

Enumerations

- `[0 .. 5]` $\rightsquigarrow$ `[0 1 2 3 4 5]`
- `[0 2 .. 5]` $\rightsquigarrow$ `[0 2 4]`
- `[5 -- 0]` $\rightsquigarrow$ `[5 4 3 2 1 0]`

Enumerations

- `[0 .. 5]` $\rightsquigarrow$ `[0 1 2 3 4 5]`
- `[0 2 .. 5]` $\rightsquigarrow$ `[0 2 4]`
- `[5 -- 0]` $\rightsquigarrow$ `[5 4 3 2 1 0]`
- `[0 .. ]` $\rightsquigarrow$ an "infinite" list

Enumerations

- `[0 .. 5]` $\rightsquigarrow$ `[0 1 2 3 4 5]`
- `[0 2 .. 5]` $\rightsquigarrow$ `[0 2 4]`
- `[5 -- 0]` $\rightsquigarrow$ `[5 4 3 2 1 0]`
- `[0 .. ]` $\rightsquigarrow$ an "infinite" list

Values in infinite lists depend on word widths:

Enumerations

- `[0 .. 5]` $\rightsquigarrow$ `[0 1 2 3 4 5]`
- `[0 2 .. 5]` $\rightsquigarrow$ `[0 2 4]`
- `[5 -- 0]` $\rightsquigarrow$ `[5 4 3 2 1 0]`
- `[0 .. ]` $\rightsquigarrow$ an "infinite" list

Values in infinite lists depend on word widths:

- `[0 .. ] : [inf][1]` $\rightsquigarrow$ `[0 1 0 1 0 1 0` ...

Enumerations

- `[0 .. 5]` $\rightsquigarrow$ `[0 1 2 3 4 5]`
- `[0 2 .. 5]` $\rightsquigarrow$ `[0 2 4]`
- `[5 -- 0]` $\rightsquigarrow$ `[5 4 3 2 1 0]`
- `[0 .. ]` $\rightsquigarrow$ an "infinite" list

Values in infinite lists depend on word widths:

- `[0 .. ] : [inf][1]` $\rightsquigarrow$ `[0 1 0 1 0 1 0` ...
- `[0 .. ] : [inf][2]` $\rightsquigarrow$ `[0 1 2 3 0 1 2` ...

Enumerations

- `[0 .. 5]` $\rightsquigarrow$ `[0 1 2 3 4 5]`
- `[0 2 .. 5]` $\rightsquigarrow$ `[0 2 4]`
- `[5 -- 0]` $\rightsquigarrow$ `[5 4 3 2 1 0]`
- `[0 .. ]` $\rightsquigarrow$ an "infinite" list

Values in infinite lists depend on word widths:

- `[0 .. ] : [inf][1]` $\rightsquigarrow$ `[0 1 0 1 0 1 0` ...
- `[0 .. ] : [inf][2]` $\rightsquigarrow$ `[0 1 2 3 0 1 2` ...
- `[0 3 .. ] : [inf][3]` $\rightsquigarrow$ `[0 3 6 1 4 7 2` ...

Sequence Comprehensions

- Cartesian:

```
[| x ^ y || x <- [0 .. 1], y <- [1 .. 3] |]
```

$\rightsquigarrow$

```
[1 2 3 0 3 2]
```

Sequence Comprehensions

- Cartesian:

```
[| x ^ y || x <- [0 .. 1], y <- [1 .. 3] |]
```

⇝

```
[1 2 3 0 3 2]
```

- Parallel:

```
[| x ^ y || x <- [0 .. 1] || y <- [1 .. 3] |]
```

⇝

```
[1 3]
```

Sequence Comprehensions

- Cartesian:

```
[| x ^ y || x <- [0 .. 1], y <- [1 .. 3] |]
```

$\rightsquigarrow$

```
[1 2 3 0 3 2]
```

- Parallel:

```
[| x ^ y || x <- [0 .. 1] || y <- [1 .. 3] |]
```

$\rightsquigarrow$

```
[1 3]
```

- Self referential:

```
fibs: [inf][32];
fibs = [0 1] # [| x + y || x <- fibs
                        || y <- tail (fibs) |];
```

```
fibs
```

$\rightsquigarrow$

Sequence Comprehensions

- Cartesian:

```
[| x ^ y || x <- [0 .. 1], y <- [1 .. 3] |]
```

$\rightsquigarrow$

```
[1 2 3 0 3 2]
```

- Parallel:

```
[| x ^ y || x <- [0 .. 1] || y <- [1 .. 3] |]
```

$\rightsquigarrow$

```
[1 3]
```

- Self referential:

```
fibs: [inf][32];
fibs = [0 1] # [| x + y || x <- fibs
                        || y <- tail (fibs) |];
```

```
fibs
```

$\rightsquigarrow$

Sequence Comprehensions

- Cartesian:

```
[| x ^ y || x <- [0 .. 1], y <- [1 .. 3] |]
```

⇝

```
[1 2 3 0 3 2]
```

- Parallel:

```
[| x ^ y || x <- [0 .. 1] || y <- [1 .. 3] |]
```

⇝

```
[1 3]
```

- Self referential:

```
fibs: [inf][32];
fibs = [0 1] # [| x + y || x <- fibs
                        || y <- tail (fibs) |];
```

```
fibs
```

⇝

```
[0 1 1 2 3 5 8
```

...

Sequence Comprehensions

- Cartesian:

```
[| x ^ y || x <- [0 .. 1], y <- [1 .. 3] |]
```

⇝

```
[1 2 3 0 3 2]
```

- Parallel:

```
[| x ^ y || x <- [0 .. 1] || y <- [1 .. 3] |]
```

⇝

```
[1 3]
```

- Self referential[3]:

```
fibs: [inf][32];
fibs = [0 1] # [| x + y || x <- fibs
                        || y <- tail (fibs) |];
```

```
fibs
```

⇝

```
[0 1 1 2 3 5 8
```

...

[3] Mathematicians might prefer "recurrence relations".

Control Structures

- Cryptol's if-then-else is like C's ternary operator ? :.

Control Structures

- Cryptol's if-then-else is like C's ternary operator ? :.
- Sequence comprehensions may be thought of as for loops. For example, in C, $n!$ is the value of `f` after executing:

```
f = 1;
for (i = 1; n >= i; i++)
  f *= i;
```

Control Structures

- Cryptol's if-then-else is like C's ternary operator `? :`.
- Sequence comprehensions may be thought of as for loops. For example, in C, $n!$ is the value of `f` after executing:

```
f = 1;
for (i = 1; n >= i; i++)
  f *= i;
```

In Cryptol, $n!$ is the value of `fs @ n` given:

```
fs : [inf][32];
fs = [1] # [| f * i || f <- fs || i <- [1 ..] |];
```

Control Structures

- Cryptol's if-then-else is like C's ternary operator `? :`.
- Sequence comprehensions may be thought of as for loops. For example, in C, $n!$ is the value of `f` after executing:

```
f = 1;
for (i = 1; n >= i; i++)
  f *= i;
```

In Cryptol, $n!$ is the value of `fs @ n` given:

```
fs : [inf][32];
fs = [1] # [| f * i || f <- fs || i <- [1 ..] |];
```

- Recursion in Cryptol gives an equivalent formulation of while loops, but that is rarely needed in cryptographic algorithms.