Cryptol Reference Card

A brief summary of Cryptol's syntax and basic patterns. For more details, look for the book *Programming Cryptol*.

Starting Cryptol, basic commands

Start Cryptol either by itself, or with the name of a Cryptol file as its argument:

```
% cryptol myfile.cry
```

```
                        _        _
   ___ _ __ _   _ _ __ | |_ ___ | |
  / __| '__| | | | '_ \| __/ _ \| |
 | (__| |  | |_| | |_) | || (_) | |
  \___|_|   \__, | .__/ \__\___/|_|
            |___/|_|  version 2.4.0

Loading module Cryptol
Cryptol>
```

At the Cryptol prompt, a number of commands are available:

```
:set base=10
:set ascii=on // or =off
:r  // or :reload - reloads file
:l <filename> // or :load <filename>
:e // edits the current file
```

Pressing Tab at the `Cryptol>` prompt, after typing `:`, or after `:set` shows you available options. Experiment!

Data and types

Cryptol's types include:

- *Bit* values can be `True` or `False`
- *Sequences* whose elements must have the same type, like `[0xa, 0xb, 0xc]`. Elements of sequences can be accessed with the `@` and `!` indexing operators, which index from the beginning and the end of the sequence, respectively. For example `[0xa, 0xb, 0xc]@0` is `0xa`.
- *Tuples* whose elements can have different types, like `('a', 0xf00d, (True, False))` Elements of tuples can be accessed via `.` like this: `(True, 0xa).0` is `True`.
- *Records* are like tuples with named elements, and are also accessed with `.` like this: `{ foo=1, bar=2 }.bar` is 2.

Querying types

Functions, expressions and values all have types in Cryptol. You can ask Cryptol what the type of something is with the `:t` command, like this:

```
Cryptol> :t "Hello, world!"
"Hello, world!" : [13][8]
Cryptol> :t 123
123 : {a} (fin a, a >= 7) => [a]
Cryptol> :t 0xf00d
0xf00d : [16]
Cryptol> :t (||)
(||) : {a} a -> a -> a
Cryptol> :t (+)
(+) : {a} (Arith a) => a -> a -> a
```

These say, in order:

- "the string 'Hello, world!' is a sequence of 13 8-bit words
- the number 123 takes a finite number, a, of bits, where $a >= 7$.
- the number `0xf00d` is a 16-bit sequence
- the operator `||` takes two arguments of the same type, and its return value has the same type
- the operator `+` takes two arguments of class `Arith` and returns a value of the same type

Sequence constructors

Sequences are the most commonly-used structure in Cryptol. They can be constructed in a number of ways. We've already seen sequence literals, like `[1, 2, 3]`. You can construct sequences of finite length by specifying the first and last elements, like this:

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

Specify the step between elements by providing the first two:

```
Cryptol> [1,3 .. 10]
Assuming a = 4
[1, 3, 5, 7, 9]
```

Descending sequences have descending first two elements:

```
Cryptol> [10, 9 .. 0]
Assuming a = 4
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Infinite sequences are created with `...` Since the elements of a sequence all have to have the same (finite) type, the sequence will wrap around. For example:

```
Cryptol> [1, 2 ...] // 2 bits per element
Assuming a = 2
[1, 2, 3, 0, 1, ...]
Cryptol> [0x1, 0x3 ...] // 4 bits/elt
[1, 3, 5, 7, 9, ...]
```

Finally, Cryptol lets you specify the elements of a sequence with the *sequence comprehension* syntax, which is inspired by the same concept in mathematics notation. Here are some examples:

```
Cryptol> [ a + 3 | a <- [1 .. 10] ]
[4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
Cryptol> [ a + b | a <- [1 .. 10] \
                 | b <- [3 ...] ]
[4, 6, 8, 10, 12, 14, 0, 2, 4, 6]
```

The first one would be described as "a sequence of elements whose values are $a + 3$, where a is drawn from the sequence '1 to 10' ". The second one is a *parallel comprehension*. The resulting length of a parallel comprehension is the shortest length of its *legs*. In this case, we would describe it as "a sequence of elements whose values are $a + b$ where a is drawn from the sequence '1 to 10' and b is drawn from the infinite sequence starting at 3".

Cryptol also supports *Cartesian comprehensions*, whose legs are separated by commas. The first element of the first leg is paired with each element of the other leg(s), followed by the second element of the first leg, etc:

```
Cryptol> [(a, b) | a <- [1, 2]\
                 , b <- [3, 4] ]
[(1, 3), (1, 4), (2, 3), (2, 4)]
```

Other things you can do with sequences: the built-in `reverse` function reverses the elements of a sequence,

and the # operator concatenates two sequences (of course the two arguments to # must have elements of the same type).

```
Cryptol> reverse ['0' .. '9'] # ['a' .. 'z']
"9876543210abcdefghijklmnopqrstuvwxyz"
```

Functions in Cryptol

A function declaration looks like this:

```
f x y = x + y
```

The type of this function is:

```
f : {a} (Arith a) => a -> a -> a
```

Reading the type of a function in Cryptol, you have:

- the name of the function,
- `:`
- in curly braces, `{}`, the names of *type variables*,
- in parentheses the *type constraints* (in this case, that the type `a` needs to be something that supports arithmetic operations (`Arith`),
- a *fat arrow* (`=>`),
- the types of the arguments, separated by `->`
- the final element is the type of the return value.

Cryptol supports *curried* arguments, which means that if you pass fewer arguments than a function takes, the result is a function that accepts the remaining arguments and has the same return type. For example:

```
addThree = f 3
addThree 8 // value is 11
```

You can provide the type of a function along with its definition, or you can let Cryptol infer a function's type for you. Sometimes providing the type can help Cryptol give you better error messages - Cryptol's inferred types are very general, and can be confusing sometimes.

Type variables

Type variables can occur in a few contexts in a Cryptol program. Some functions take type-level arguments, such as `take` and `drop`:

```
take`{2}[1 .. 10]
// value: [1, 2]
drop`{2}[1 .. 10]
// value: [3, 4 .. 10]
```

Type-level arguments can be positional or named, like this:

```
Cryptol> :t take
take : {front, back, elem} (fin front) =>
  [front + back]elem -> [front]elem
Cryptol> take`{back=3}[1 .. 10]
[1, 2, 3, 4, 5, 6, 7]
```

We asked for the type of `take`, and we learned that its argument is *front* + *back* elements long, and that it returns *front* elements. If we pass `back=3` as the type argument to `take`, Cryptol knows the list has 10 elements, and also that $front + 3 = 10$, so it infers that we want to take 7 elements of our list `[1 .. 10]`.

`:prove` and `:sat`

If a function returns `Bit`, you can use Cryptol's powerful theorem proving powers to either prove that the function is true *for all* values of its argument(s), using `:prove`, or that *there exists* at least one input value that makes the function return True, using `:sat`. Here, we prove that $x+x$ is always equal to $x << 1$.

```
Cryptol> :prove (\x -> x + x == x << 1):[32]->Bit
Q.E.D.
```

One limitation of `:prove` and `:sat` is that the arguments have to be *monomorphic*. That's why we had to specify the function had type `[32]->Bit`. Here we prove that $x + y == y + x$ for all 32-bit values x and y:

```
Main> :prove (\x y -> x+y == y + x):[32]->[32]->Bit
Q.E.D.
```

Here we ask Cryptol if there are an 8-bit x and y such that $x^y = 123$:

```
Cryptol> :sat (\x y -> x ^^ y == 123):[8]->[8]->Bit
((\x y -> x ^^ y == 123) : [8] -> [8] -> Bit)
51 31 = True
Cryptol> 51 ^^ 31:[8]
123
```

Defining types

Typically all type names start with capital letters. Define a new type like this:

```
type NumChars = 26
type Permutation=[NumChars]Char
type Index=[7]
```

Having defined this type, you can define functions that take arguments, or return values of that type. Your code becomes more readable with descriptive type names.

```
invertPermutation : Permutation -> Permutation
invertPermutation perm =
    [ indexToChar (indexOf c perm)
    | c <- alphabet
    ]
indexToChar : Index -> Char
indexToChar i = (0b0 # i) + 'A'
```

Cryptol idioms

Pretty much everything you do in Cryptol should be operating on sequences. If you're tempted to call a function recursively, you'll probably be better served by refactoring your problem into recursive operations on sequences. Here's an idiomatic Fibonacci sequence in Cryptol:

```
fibs = [0,1] #
    [ a + b
    | a <- fibs
    | b <- drop`{1}fibs]
take`{10}(fibs:[_][12])
// value: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
(fibs:[_][64])@85
// value: 0x039a9fadb327f099
```

This reference card is by Dylan McNamee, and is hosted on GitHub. See `github.com/dylanmc/CryptoBook`