

# Cryptol version 2 Syntax

## Contents

Layout	2
Comments	3
Identifiers	3
Keywords and Built-in Operators	3
Numeric Literals	4
Bits	5
If Then Else with Multiway	5
Tuples and Records	5
Sequences	7
Functions	7
Local Declarations	8
Explicit Type Instantiation	8
Demoting Numeric Types to Values	8
Explicit Type Annotations	8
Type Signatures	9
Type Synonym Declarations	9
Modules	9
Hierarchical Module Names	9

<b>Module Imports</b>	<b>9</b>
<b>Import Lists</b>	<b>10</b>
<b>Hiding Imports</b>	<b>10</b>
<b>Qualified Module Imports</b>	<b>11</b>
<b>Private Blocks</b>	<b>12</b>
<b>Parameterized Modules</b>	<b>12</b>
<b>Named Module Instantiations</b>	<b>13</b>
<b>Parameterized Instantiations</b>	<b>14</b>
<b>Importing Parameterized Modules</b>	<b>14</b>

## 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**.

## 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 */
```

## 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
```

## Keywords and Built-in Operators

The following identifiers have special meanings in Cryptol, and may not be used for programmer defined names:

Arith	Inf	export	import	min	property	width
Bit	SignedCmp	extern	inf	module	then	
Cmp	True	fin	lg2	newtype	type	
False	else	if	max	pragma	where	

The following table contains Cryptol's operators and their associativity with lowest precedence operators first, and highest precedence last.

Table 1: Operator precedences.

Operator	Associativity
<code>==&gt;</code>	right
<code>\/</code>	right
<code>/\</code>	right
<code>-&gt;</code> (types)	right

Operator	Associativity
!= ==	not associative
> < <= >= <\$ >\$ <=\$ >=\$	not associative
	right
^	left
&&	right
#	right
>> << >>> <<< >>=\$	left
+ -	left
* / % /\$ %\$	left
^^	right
! !! @ @@	left
(unary) - ~	right

## 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 represent finite bit sequences (i.e., they have type `[n]`). Using binary, octal, and hexadecimal notation results in bit sequences of a fixed length, depending on the number of digits in the literal. Decimal literals are overloaded, and so the length of the sequence 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           // : {n}. (fin n, n >= 4) => [n]
            // (need at least 4 bits)

0           // : {n}. (fin n) => [n]
```

## 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 2: Bit operations.

Operator	Associativity	Description
<code>==&gt;</code>	right	Short-cut implication
<code>\ </code>	right	Short-cut or
<code>/\</code>	right	Short-cut and
<code>! = ==</code>	none	Not equals, equals
<code>&gt; &lt; &lt;= &gt;= &lt;\$ &gt;\$ &lt;=\$ &gt;=\$</code>	none	Comparisons
<code>  </code>	right	Logical or
<code>^</code>	left	Exclusive-or
<code>&amp;&amp;</code>	right	Logical and
<code>~</code>	right	Logical negation

## If Then Else with Multiway

`If then else` has been extended to support multi-way conditionals. Examples:

```
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
```

## Tuples and Records

Tuples and records are used for packaging multiples values together. Tuples are enclosed in parenthesis, while records are enclosed in 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
```

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
```

## 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,  $[n] \text{ Bit}$ , is called a *word*. We may abbreviate the type  $[n] \text{ Bit}$  as  $[n]$ . An infinite sequence with elements of type  $a$  has type  $[\text{inf}] a$ , and  $[\text{inf}]$  is an infinite stream of bits.

```
[e1,e2,e3]                // A sequence with three elements

[t .. ]                  // Sequence enumerations
[t1, t2 .. ]            // Step by t2 - t1
[t1 .. t3 ]
[t1, t2 .. t3 ]
[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 ]
```

Note: the bounds in finite unbounded (those with  $..$ ) sequences are type expressions, while the bounds in bounded-finite and infinite sequences are value expressions.

Table 3: 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 point-wise operations.

```
[p1, p2, p3, p4]        // Sequence pattern
p1 # p2                  // Split sequence pattern
```

## Functions

```
\p1 p2 -> e             // Lambda expression
```

```
f p1 p2 = e           // Function definition
```

## 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.

## Explicit Type Instantiation

If `f` is a polymorphic value with type:

```
f : { tyParam }  
f = zero
```

you can evaluate `f`, passing it a type parameter:

```
f `{ tyParam = 13 }
```

## 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} : {w >= width t}. [w]
```

## Explicit Type Annotations

Explicit type annotations may be added on expressions, patterns, and in argument definitions.

```
e : t
```

```
p : t
```

```
f (x : t) = ...
```



## Type Signatures

```
f,g : {a,b} (fin a) => [a] b
```

## Type Synonym Declarations

```
type T a b = [a] b
```

## Modules

A *module* is used to group some related definitions.

```
module M where
```

```
type T = [8]
```

```
f : [8]
```

```
f = 10
```

## 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`.

## 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`

```

## 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.

## 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
```

## 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 to use the same qualifier prefix for imports from different modules. For example:

```
import A as B  
import X as B
```

Such declarations will introduce all definitions from A and X but to use them, you would have to qualify using the prefix `B::`.

## 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
```

## 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

```

## 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`.

## 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`.

## 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.