Cryptol version 2 Syntax

Contents

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:

```
else      include   property   let      infixl      parameter
extern    module    then       import   infixr      constraint
if        newtype   type       as       infix
private   pragma    where      hiding   primitive
```

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
`==>`	right
`\/`	right

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

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 2: Type-level operators

Operator	Meaning
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`/^`	Ceiling division (`/` rounded up)
`%`	Modulus
`%^`	Ceiling modulus (compute padding)
`^^`	Exponentiation
`lg2`	Ceiling logarithm (base 2)
`width`	Bit width (equal to `lg2(n+1)`)

Operator	Meaning
`max`	Maximum
`min`	Minimum

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 closest representable 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
```

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 ...`
```

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 3: Bit operations.

Operator	Associativity	Description
`==>`	right	Short-cut implication
`\/`	right	Short-cut or
`/\`	right	Short-cut and
`!= ==`	none	Not equals, equals
`> < <= >= <$ >$ <=$ >=$`	none	Comparisons

Operator	Associativity	Description
`||`	right	Logical or
`^`	left	Exclusive-or
`&&`	right	Logical and
`~`	right	Logical negation

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
```

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 for most purposes. 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
```

Ordering on tuples and records is defined lexicographically. Tuple components are compared in the order they appear, whereas record fields are compared in alphabetical order of field names.

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.

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 }
```

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 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
```

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 } 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 : {n} (fin n, n >= width t) => [n]
```

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 Synonyms and Newtypes

Type synonyms

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

A `type` declaration creates a synonym for a pre-existing type expression, which may optionally have arguments. A type synonym is transparently unfolded at use sites and is treated as though the user had instead written the body of the type synonym in line. Type synonyms may mention other synonyms, but it is not allowed to create a recursive collection of type synonyms.

Newtypes

```
newtype NewT a b = { seq : [a]b }
```

A `newtype` declaration declares a new named type which is defined by a record body. Unlike type synonyms, each named `newtype` is treated as a distinct type by the type checker, even if they have the same bodies. Moreover, types created by a `newtype` declaration will not be members of any typeclasses, even if the record defining their body would be. For the purposes of typechecking, two newtypes are considered equal only if all their arguments are equal, even if the arguments do not appear in the body of the newtype, or are otherwise irrelevant. Just like type synonyms, newtypes are not allowed to form recursive groups.

Every `newtype` declaration brings into scope a new function with the same name as the type which can be used to create values of the newtype.

```
x : NewT 3 Integer
x = NewT { seq = [1,2,3] }
```

Just as with records, field projections can be used directly on values of newtypes to extract the values in the body of the type.

```
> sum x.seq
6
```

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
```

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 qualifier `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 introduces 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` introduces 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 instantiation 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.