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 declaration, one for `f` and one for `g`. All the lines between `f` and `g` that are indented more then `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       extern    inf    module      then
Bit     True      fin       lg2    newtype     type
Cmp     else      if        max    pragma      where
False   export    import    min    property    width
```

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
`||`	left
`^`	left
`&&`	left
`->` (types)	right
`!= ==`	not associative
`> < <= >=`	not associative
`#`	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
`||`	left	Logical or
`^`	left	Exclusive-or
`&&`	left	Logical and
`!= ==`	none	Not equals, equals
`> < <= >=`	none	Comparisons
`~`	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 parenthesis, 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] 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

[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)
`@ !`	Access elements (front,back)
`@@ !!`	Access sub-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
```