

October, 2008
Version 1.8



Programming Guide

Cryptol:

The Language of Cryptography

| galois |

Galois, Inc.
421 SW 6th Avenue | Suite 300 | Portland, OR 97204
T 503.626.6616 | F 503.350.0833
www.galois.com

IMPORTANT NOTICE

This documentation is furnished for informational use only and is subject to change without notice. Galois, Inc. assumes no responsibility or liability for any errors or inaccuracies that may appear in this documentation. [Note: This paragraph is NOT included for purposes of copyright protection. It is a liability reduction paragraph, and falls outside the scope of this document. This paragraph should be reviewed by qualified legal counsel to determine its appropriateness.]

Copyright 2003-2008 Galois, Inc. All rights reserved by Galois, Inc.

This documentation, or any part of it, may not be reproduced, displayed, photocopied, transmitted, or otherwise copied in any form or by any means now known or later developed, such as electronic, optical, or mechanical means, without express written authorization from Galois, Inc.

Warning: This computer program and its documentation are protected by copyright law and international treaties. Any unauthorized copying or distribution of this program, its documentation, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted under the maximum extent possible under the law.

The software installed in accordance with this documentation is copyrighted and licensed by Galois, Inc. under separate license agreement. This software may only be used pursuant to the terms and conditions of such license agreement. Any other use may be a violation of law.

TRADEMARKS

Cryptol is a registered trademark of Galois, Inc. in the United States and other countries.

UNIX is a registered trademark of The Open Group in the U. S. and other countries.

Linux is a registered trademark of Linus Torvalds.

Solaris, Java, Java Runtime Edition, JRE, and other Java-related marks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

XILINX, the Xilinx logo, and other designated brands included herein are trademarks of Xilinx, Inc.

Other company or product names mentioned herein may be trademarks or registered trademarks of their respective owners. Trademark specifications are subject to change without notice. All terms mentioned in this documentation that are known to be trademarks or service marks have been appropriately capitalized to the best of our knowledge; however, Galois cannot attest to the accuracy of all trademark information. Use of a term in this documentation should not be regarded as affecting the validity of any trademark or service mark.

Galois, Inc

421 SW Sixth Avenue, Suite 300
Portland, OR 97204

Table of Contents

	List of Tables	v
	List of Examples	vii
	Preface	ix
SECTION 1	Introduction	1
	1.1 High-level view of Cryptol	3
	1.2 Definitions	4
SECTION 2	The Cryptol Type System	5
	2.1 Bits	6
	2.2 Numbers	7
	2.3 Sequences	8
	2.4 Tuples	10
	2.5 Sizes	11
	2.6 Shapes	12
	2.7 Controlling Polymorphism	13
	2.8 Type Annotations	14
	2.9 Defaulting	15
	2.10 Records	16
	2.11 Type synonyms	21
SECTION 3	Cryptol Language Elements	23
	3.1 Comments	23
	3.2 Literate Cryptol	24
	3.3 Arithmetic	26
	3.4 Polynomial Arithmetic	27
	3.5 Boolean operations	28
	3.6 Equality and Comparisons	29
	3.7 Conditional Expressions	30
	3.8 Shifts and Rotates	31
	3.9 Appending Sequences	32
	3.10 Joining and Splitting Sequences	33
	3.11 Sequence Comprehensions	34
	3.12 Streams and Recursively Defined Sequences	35
	3.13 Type Synonyms	36
	3.14 Pattern Bindings	41

SECTION 4	An Extended Example	43
	4.1 Preliminaries	44
	4.2 The Cipher Rounds	46
	4.3 The Round function	47
	4.4 The Key Schedule	50
SECTION 5	Predefined Functions and Values in Cryptol	53
	5.1 Arithmetic Operators	54
	5.2 Polynomial Operators	58
	5.3 Logical Operators	59
	5.4 Operators for Shifting and Rotating	63
	5.5 Operators for Sequence Manipulation	66
	5.6 Comparison Operators	72
	5.7 Miscellaneous Operators	74
	5.8 Format: Pretty printing	77
SECTION 6	Cryptol Idioms	83
	6.1 Padding	84
	6.2 For-Loops and Recurrence Relations	85
	6.3 Cryptographic Modes	86
SECTION 7	Cryptol Tips and Pitfalls	87
	7.1 Understanding the type of an enumeration	88
SECTION 8	Examples	91
	8.1 DES	92
	8.2 RC5	93
	8.3 Advanced Encryption Standard	95
	Index	99



List of Tables

Table 5-1 Format specifiers for pretty printing.

77

List of Examples

example 1-1	A simple Cryptol program.	4
example 1-2	Use of where clauses.	4
example 2-1	Indexing sequences	9
example 2-2	Multiple indexing	9
example 2-3	Indexing sequences from the end	9
example 2-4	Multiple indexing from the end	9
example 2-5	Indexing bits of a word	9
example 2-6	Rearranging elements of a sequence	12
example 2-7	Creating a record.	17
example 2-8	Specifying type signatures with field definitions.	17
example 2-9	Nested records.	17
example 2-10	Record labels must be unique at any given level.	17
example 2-11	Field selection.	18
example 2-12	Projecting fields.	18
example 2-13	Using record patterns to reduce the need to project fields.	18
example 2-14	Field names and variable names.	18
example 3-1	Using Comments	23
example 3-2	Element-wise arithmetic	26
example 3-3	Equality tests	29
example 3-4	Using if-then-else	30
example 3-5	Shifts and rotates	31
example 3-6	Using join and split	33
example 3-7	Simple sequence comprehensions	34
example 5-1	Using +	54
example 5-2	Using -	54
example 5-3	Using *	55
example 5-4	Using /	55
example 5-5	Using %	56
example 5-6	Using **	56
example 5-7	Using lg2	57
example 5-8	Using False	59
example 5-9	Using True	59
example 5-10	Using &	60
example 5-11	Using	60
example 5-12	Using ^	61
example 5-13	Using ~	61
example 5-14	Using negate	62
example 5-15	Using width	66
example 5-16	Using @	66
example 5-17	Using @@	67
example 5-18	Using #	67
example 5-19	Using join	68

example 5-20	Using splitBy	69
example 5-21	Using groupBy	69
example 5-22	Using take	69
example 5-23	Using drop	70
example 5-24	Using tail	71
example 5-25	Using reverse	71
example 5-26	Using transpose	71
example 5-27	Using comparison operators	73
example 5-28	Using zero	74
example 5-29	Using min	74
example 5-30	Using parity	75
example 5-31	Using error	75
example 5-32	Using undefined	75
example 6-1	For loops (I)	85
example 6-2	For loops (II)	85
example 7-1	Replacing the ith element of a sequence	88
example 8-1	DES Encryption	92
example 8-2	RC5 cipher	93
example 8-3	AES encryption	95



Preface

- Getting More Information

More information on Cryptol is available at the Cryptol Web site, *<http://www.cryptol.net>*.

More information on Galois Connections is available at the Galois Connections Inc. Web site, *<http://www.galois.com>*.

1

Introduction

Cryptographic components are increasingly being integrated into hardware and software systems to improve information assurance and security. Because of this, several serious challenges arise: test and verification of systems incorporating cryptography, unambiguous specification of cryptographic algorithms, and the rapid and safe retargeting of cryptographic implementations to new hardware and software platforms.

Cryptol brings a new, formal methods-based approach to cryptography that addresses these challenges. It is a high-level specification language for cryptography that was designed at Galois Connections in consultation with expert cryptographers from the National Security Agency. In Cryptol, cryptographic concepts are expressed directly and formally and in a fashion that is independent of the details of a particular hardware platform.

Cryptol provides significant benefits to:

- ◆ Crypto developers targeting a variety of hardware and software platforms
- ◆ High-assurance systems developers incorporating embedded cryptographic components
- ◆ Cryptographers exploring new algorithms
- ◆ Verification laboratories using formal models to verify implementations
- ◆ Customers of high-assurance systems that validation and test implementations

These benefits are a result of one's ability to view a single Cryptol specification from a number of perspectives. First Cryptol can be seen as a *language for Cryptography*. Using high-level Cryptol to express the same concepts and idioms as those found in published algorithms, developers can quickly implement pre-existing algorithms or develop new ones. Developers are thereby freed to focus on the cryptography itself, not distracted by machine-level details such as word size. In a complementary way, Cryptol can be seen as providing an *authoritative reference for validation*. To this end, Cryptol is positioned to become the standard language for cryptography. A growing number of both public and non-public algorithms are under development. Standard Cryptol specifications can be used to validate new cryptographic implementations by generating test vectors of user-selectable

intermediate values. Taking this line of thinking a bit further, Cryptol may also be viewed as a *framework for verification*. For embedded systems in particular, and for developers of high assurance applications in general, Cryptol facilitates construction of formal models, providing for an increased level of confidence in the development. Lastly Cryptol provides an exciting *platform for implementation generation*. In this regard it should be stated that Cryptol specifications are inherently portable. Retargeting the deployment platform does not involve recoding the algorithm. Cryptol is intended for use with various platforms, including embedded systems, smart cards, and FPGAs.

1.1 High-level view of Cryptol

Cryptography can be seen as the mathematical manipulation of sequences of data. This is reflected in the design of Cryptol: it is a language of various arithmetics glued together by a rich language of sequence manipulation.

Sequences run through all layers of a cryptographic protocol: from the sequences of bits that make up a computer word, to sequences of words of various sizes that make up the blocks of data processed by an algorithm, to blocks of data streaming through an algorithm under control of various cryptographic modes such as Cipher Block Chaining.

Sequences in Cryptol can be either finite, as in a block of words, or infinite, as in a recurrence relation, or stream cipher. In Cryptol, words, blocks, and streams are all treated uniformly as (potentially nested) sequences. This approach completely avoids the vagaries of platform-dependent word sizes. If the algorithm calls for 57-bit words, then the Cryptol specification uses 57-bit words, and programmer need not worry about how to implement that on, for example, a 32-bit architecture. Further, in the specification, the same language is used for 57-bit words as for 32-bit words, i.e. there is no need for awkward macros to work around the artificial boundaries that are found in the standard programming languages that most algorithms are written in. Not only does this make it easier to write the specification, it also makes it easier to change the algorithm, and to experiment with different word or block sizes.

This uniformity in the data model leads to a uniformity in the primitives that manipulate sequences. The same primitives that are used to manipulate words (sequences of bits) also work to manipulate blocks, and sequences of blocks, and streams. There are primitives for splitting up and joining sequences so that it is easy to employ many views of the same sequence. For example, a 32-bit word is easily split into 4 bytes, and using the same primitive, those 4 bytes are easily split into 2 4-bit nibbles. Just as easily, this structure can be collapsed back to a flat 32-bit word. Thus, in Cryptol, it is easy to view data with deeply nested structure, and with none at all, depending on what is appropriate for the algorithm.

Data model uniformity leads to uniformity in the control structures of the language: sequences are constructed and iterated over using *sequence comprehensions*, inspired by mathematical set comprehensions. Control at all levels, from bits to streams of blocks, is expressed using sequence comprehensions. Combining sequence comprehensions with recursive definitions allows us to directly express sophisticated recurrence relations that appear often in cryptographic algorithms.

The complement the powerful data model, Cryptol also has a sophisticated type system. The type system keeps track of the sizes of words and sequences, and ensures that incompatible data isn't accidentally mixed—a classic source of confusion. The type system also enhances the declarative nature of the language by allowing tedious details like correct padding to be specified very simply using type annotations, leaving the gritty details to be sorted out automatically by Cryptol.

1.2 Definitions

A Cryptol program consists of a collection of definitions. This contrasts with a typical imperative language like C, which is a sequence of assignments. Thus, the order of definitions is unimportant, and there are no side effects to a definition. Avoiding side effects means that a specification can be understood by understanding each of its parts, without having to understand its context.

Example 1-1 declares three names:

- ◆ A constant `x`, whose value is bound to 13,
- ◆ A function `incr`, which increments its argument by 1. Notice that the function does *not* change the value of its argument, it simply returns its argument's value incremented by 1.
- ◆ Another function `foo`, which takes two arguments and performs a simple mathematical computation on its arguments.

EXAMPLE 1-1 *A simple Cryptol program.*

```
x = 13;
incr x = x + 1;
foo (x, y) = 2 * x + 3 * y - 1;
```

Definitions can also include local definitions, specified in a `where` clause. In Example 1-2, function `f` defines two local values named `a` and `b` that are only visible within `f`. The subsequent local binding to `a` in the definition of `y` is completely separate. Note that there can be at most one binding to a name within a given `where` clause.

EXAMPLE 1-2 *Use of where clauses.*

```
f x = a + b
  where { a = x * 2;
          b = x - 1;
        };

y = 3 * a
  where a = 0x1234;
```

2

The Cryptol Type System

2.1 Bits

The building blocks of Cryptol are bits, which are written `True` for a one bit, and `False` for a zero bit. The more common elements of Cryptol programs are vectors of bits, referred to as words. The type of a bit is written as `Bit`.

2.2 Numbers

Numeric literals in Cryptol are represented as words. The size of the vector are at least the size necessary to represent the literal encoded as bits. The type of a word is written by enclosing the size in bits in square brackets: e.g. the type of a 32-bit word is written: [32]. In the body of a program, type declarations look like:

```
w : [17];  
w = 18;
```

This declares that `w` is a name for the number 18, and it is stored in a 17-bit word. This example illustrates the simplest form of a type declaration. The programmer does not have to give types for entries; Cryptol will infer them automatically. But it is considered good style to write down explicit types. We will investigate types in detail later. Also notice that words can be of any size, and, in particular, that their size is not constrained by the underlying machine implementation.

Numeric literals can either be written in the usual base 10, or expressed as hexadecimal, octal, or binary constants, by prefixing with `0x`, `0o`, or `0b` respectively. For example, `0x1fc3` is the hexadecimal constant 1fc3.

2.3 Sequences

The primary way of structuring data in Cryptol is via sequences. Sequences are ordered collections of data, all of which are the same size and type. The elements of a sequence may be bits, or other sequences. A word is just a sequence of bits.

Sequences are written as square brackets around space-separated elements. For example, the following specifies a sequence consisting of ten words:

```
[1 2 3 4 5 6 7 8 9 10]
```

Assuming we are working with 4-bit literals, the type for the above sequence is `[10][4]`, indicating a sequence of length 10 containing elements that are 4-bit words. In general, a sequence with elements of type *a* has type `[n]a`, where *n* is the number of elements in the sequence. We call *n* the **size** of the sequence, and we call *a* the **shape** of the sequence.

It is often convenient to specify the elements of a sequence as an enumeration. In Cryptol this is done by specifying a range of values in brackets separated by “..”. For example, the previous sequence consisting of words from 1 to 10 can also be written as follows:

```
[1 .. 10]
```

If two starting elements are provided, the difference between the two provides the increment for subsequent values. Thus:

```
[1 3 .. 10]
```

is the same as:

```
[1 3 5 7 9]
```

It is also possible to specify infinite sequences using this notation, as in `[1 ..]`. We will come back to infinite data later on.

Descending sequence enumerations may be specified using the “- -” notation. If you wanted a sequence descending from 10 to 1, you could write:

```
[10 - - 1]
```

As for ascending enumerations, if two starting elements are provided, the difference between the two provides the decrement for subsequent values. Thus:

```
[10 8 - - 1]
```

is the same as:

```
[10 8 6 4 2]
```

The elements of a sequence are indexed from left to right, starting with 0, using the operator `@`.

EXAMPLE 2-1 *Indexing sequences*

In the following, y will have value 9:

```
xs = [13 27 9 34];
y = xs @ 2;
```

The @@ operator can be used to construct a new sequence out of elements of another sequence.

EXAMPLE 2-2 *Multiple indexing*

In the following, ys will have value [27 34]:

```
xs = [13 27 9 34];
ys = xs @@ [1 3];
```

Often it is useful to index from the end of a sequence instead of the beginning. The ! and !! operators provide this.

EXAMPLE 2-3 *Indexing sequences from the end*

In the following, y will have value 9:

```
xs = [13 27 9 34];
y = xs ! 1;
```

The expression `xs ! i` is equivalent to `xs @ (width xs - i - 1)`.

EXAMPLE 2-4 *Multiple indexing from the end*

In the following, ys will have value [9 13]:

```
xs = [13 27 9 23];
ys = xs !! [1 3];
```

The bits of a word are indexed in little-endian fashion (that is, the least significant bit is indexed by 0).

EXAMPLE 2-5 *Indexing bits of a word*

In the following, y will have value False and z will have value 5.

```
x = 0xf5;
y = 0xfe @ 0;
z = x @@ [0 .. 3];
```

The choice of using a little-endian encoding in Cryptol is an arbitrary, but consistent, design choice. However, the use of a big-endian encoding is also common. As an experimental feature, the use of big-endian encoding in the bit-mode can be selected by using the +B flag. See Section 2.5, "Cryptol interpreter options and flags," on page 11 in *The Cryptol Toolset*.

2.4 Tuples

An alternate way of structuring data is to use a tuple. A tuple is an ordered collection of data, all of which can be of different types.

Tuples are written as parentheses around comma-separated elements. For example, the following is a tuple containing a bit, a word, and a sequence of bytes:

```
(True, 121, "abc")
```

The type of a tuple is written similarly. For example, a type for the above tuple might be:

```
(Bit, [12], [3][8])
```

2.5 Sizes

Cryptol has a very flexible notion of the *size* of data. For example, we can have a sequence of eight elements, each being 32 bits, this would be expressed as a type of `[8][32]`. When a numeric literal is used in a program, it will take on whatever size (i.e. number of bits) is demanded by its surrounding context. For example, if we have a function `twizzle`, which expects a 32-bit word argument, then the parameter in the call `twizzle 14` are interpreted as a 32-bit literal.

But things get interesting when the context *doesn't* immediately constrain the size. A value or function with unconstrained sizes is *size polymorphic*. However, this polymorphism isn't completely unbounded—a numeric literal is constrained to be at least as large as the number of bits necessary to represent the literal itself. For example, the literal 32 would require at least 6 bits. This constraint is reflected in the type given, with size constraints given to the left of a `=>`. Therefore, the literal 32 has the following type.

```
(a >= 6) => [a]
```

Many operations in Cryptol only make sense when applied to finite sequences, such as reversing a list. For these cases, the special type predicate `fin` may be used to indicate a type that must be finite. For example, the type of a function that reverses sequences would be as follows.

```
(fin a) => [a]b -> [a]b
```

The built-in function `width` can be used to determine the size of the outermost dimension of an expression. For example:

```
cryptol> width 32
6
cryptol> width [17 2 32 14]
4
cryptol> width [[1] [2] [3]]
3
```

2.6 Shapes

In addition to size polymorphism, Cryptol functions can also be polymorphic about the number of dimensions a value has, i.e., its shape. This is expressed in the type of a sequence by putting a type variable after the sequence of outer dimensions. For example, the type of a sequence that has four elements (when we don't know or care what those elements are) would be `[4]a`.

EXAMPLE 2-6 *Rearranging elements of a sequence*

The function `swab` rearranges the elements of a sequence of 4 elements. Notice that the type of individual elements does not matter, as indicated by the type variable `a`:

```
swab : {a} [4]a -> [4]a;  
swab [a b c d] = [b a d c];
```

2.7 Controlling Polymorphism

Definitions in Cryptol can be given a type signature. The use of the signature is optional. However, it is often useful, and sometimes even necessary: the flexibility of the type system can lead to programs that are more polymorphic than the programmer expects or needs, and type signatures can be used to restrict polymorphism. For example, the following defines a constant, which is constrained to be exactly 32 bits wide:

```
x : [32];
x = 13;
```

When writing a signature for a polymorphic value, the type variables must be declared. Type variables are declared by wrapping the collection of them in curly braces in front of the type, as in the following example:

```
f : {a} [a] -> [a];
f x = x;
```

The type of `f` is read as: for all sizes a , `f` takes an a -sized word and returns a word of the same size.

Consider the definition of `xor` given in Section 3.11, "Sequence Comprehensions," on page 34. If no type-signatures are given, Cryptol infers the following type for `xor`:

```
xor : {a b c} ([a]b,[c]b) -> [min(a,c)]b
```

A moment of thought reveals that this is the most general type for the definition of `xor`, but it may not necessarily be what the user wants. If the user had intended to define a function only over words, then this definition is both more size polymorphic (works over sequences of differing sizes) and shape polymorphic (works over arbitrarily-dimensioned sequences) than desired. While this flexibility might be useful in some cases, often it will lead to ambiguities later on. A more useful type might be to constrain it to work only on arbitrary-sized words of the same length. To achieve this, we can specify the following type for `xor`:

```
xor : {a} ([a], [a]) -> [a]
```

2.8 Type Annotations

Another way to control polymorphism is to give type annotations on expressions. Type annotations have the form of an expression followed by a colon and a type. These are useful when it isn't convenient to provide a type signature. For example, if you wish to force a literal to be 16 bits wide, you can annotate it as follows:

```
13 : [16]
```

These annotations can appear on any expression, although you should parenthesize for clarity. For example, you could annotate just one argument in a function application with a type.

```
x + (13 : [16])
```


2.9 Defaulting

Whenever a definition has free constraints (constraints of type variables that don't occur in the type) it is subject to the defaulting rule. The defaulting rule simply assigns the smallest size that will satisfy the constraint.

```
kb : {a} (a >= 4) => [a];  
kb = 8;  
foo = [1 .. 199] @ kb;
```

Here the use of kb in foo are defaulted to 4 bits, since the constraint gives that as the lower bound.

2.10 Records

When dealing with crypto-algorithms, a mechanism that packages related values and functions together, enables the succinct description and use of interfaces.

The Cryptol records feature is such a mechanism. It extends tuples with two additional features:

- ◆ Fields can be labeled.
- ◆ Polymorphic values can be stored as components.

2.10.1 The record system

In this section, we describe the records system, starting with a discussion of the various design choices we made.

- Cryptol records are lightweight.

In order to support various backends, and in particular, compilation down to FPGA's, the record design must avoid heavy run-time support requirements. Records are part of the static semantics, and are compiled away in resource constrained backends.

Note that the compile-time restriction does not prohibit the Cryptol language from treating records as values. Users may declare functions returning records, bind variables to record values, etc. That is, for all practical purposes, records are a first class citizen of the language. This particular aspect is essential for providing a basic interface definition mechanism.

However, particular backends may impose restrictions on what can be supported, following the usual Cryptol strategy. (For instance, while one can have record fields holding functions in the ``interpreted'' versions, the Cryptol FPGA compiler requires record fields to be non-functions.)

- Cryptol records are anonymous.

Users are not required to declare records. In particular, records remain anonymous, and are identified only by their field names in type signatures and record patterns (see Section 2.10.4, "Record patterns," on page 18).

While this choice can make some type signatures overly complex, the usual type-inference mechanism of Cryptol should ease any undue burden. With the new type synonym facility (described in Section 2.11, "Type synonyms," on page 21), record types can be given names as well, potentially streamlining the user experience.

2.10.2 Creating records

Records are created by giving values to fields. The syntax for record creation follows tuple-creation closely, replacing parentheses with braces, and commas with semicolon, as shown in Example 2-7 on page 17.

EXAMPLE 2-7 *Creating a record.*

```
Cryptol> {foo = True; bar = (False, True)}
{bar=(False, True); foo=True}

: {bar : (Bit, Bit); foo : Bit}
```

The order of fields (i.e., `foo` and `bar` in the above example) is immaterial. For presentation purposes, however, Cryptol will internally sort the field names in alphabetical order when printing record values and types.

The record creation syntax closely follows the syntax of the `where` clauses in Cryptol. In fact, `where` clause bodies and records are parsed in precisely the same way.*

EXAMPLE 2-8 *Specifying type signatures with field definitions.*

```
Cryptol> {foo = True; bar : [2]; bar = 0}
{bar=0x0; foo=True}

: {bar : [2]; foo : Bit}
```

EXAMPLE 2-9 *Nested records.*

```
Cryptol> {foo = True; bar = {baz = False}}
{bar={baz=False}; foo=True}

: {bar : {baz : Bit}; foo : Bit}
```

EXAMPLE 2-10 *Record labels must be unique at any given level.*

```
Cryptol> {foo = True; foo = 2}
["command line", line 1, col 2
 "command line", line 1, col 14]: Record label "foo" is multiply
defined
```

```
Cryptol> ({foo = True}, {foo = { foo = True }})
({foo=True}, {foo={foo=True}})

: ({foo : Bit}, {foo : {foo : Bit}})
```

2.10.3 Selecting fields

Field selection is performed via the ``.'` operator, as shown in Example 2-11 on page 18. The left hand side of the selection operator can be any arbitrary expression that evaluates to a record.

*.With the obvious exception that a `where` clause defining a single variable does not need braces. To create a record with a single field, the braces would still be needed.

EXAMPLE 2-11 *Field selection.*

```
Cryptol> r.foo where r = {foo = True}
True

Cryptol> {foo = True}.foo
True

Cryptol> {foo = {bar = False}}.foo.bar
False

Cryptol> (f 4).foo where f x = {foo = x}
0x4
```

2.10.4 Record patterns

One of the gripes about using patterns in practice is the constant need for projecting fields. Consider Example 2-12.

EXAMPLE 2-12 *Projecting fields.*

```
test : {a b c} (fin c) =>
  ({dec: (a,b) -> c; enc: (a,c) -> b},
   a,
   c) -> Bit;
test (alg, key, pt)
  = alg.dec (key, alg.enc (key, pt)) == pt;
```

Another disadvantage of this style is the need for providing explicit type annotations, which are required due to support for polymorphic fields. See Section 2.10.5, "Field polymorphism," on page 19 for details.

Record patterns avoid this problem; Example 2-13 demonstrates the use of record patterns to simplify the function in Example 2-12.

EXAMPLE 2-13 *Using record patterns to reduce the need to project fields.*

```
test ({dec = d; enc = e}, key, pt)
  = d (key, e (key, pt)) == pt;
```

Note that the pattern binding binds the dec field to the variable d and the enc field to e. The variables d and e can be patterns more complicated patterns themselves, although in this case, they are simple variables.

Note that Cryptol also allow punning, allowing us to use the same variable names as the record field names, as shown in Example 2-14.

EXAMPLE 2-14 *Field names and variable names.*

```
test ({dec; enc}, key, pt)
  = dec (key, enc (key, pt)) == pt;
```

Nested patterns follow the usual Cryptol pattern matching semantics. Consider the following definition:

```
pairAdd {coord1 = (x1, y1); coord2 = (x2, y2)}
  = (x1+x2, y1+y2);
```

In this case, the inferred type for `pairAdd` is:

```
Cryptol> :t pairAdd
pairAdd : {a b c d} {coord1 : ([a]b, [c]d);
coord2 : ([a]b, [c]d)} -> ([a]b, [c]d) \end{verbatim}
```

We have:

```
Cryptol> pairAdd
{coord1 = (2, 3); coord2 = (4, 6)} (0x6, 0x1)
```

(We see the usual modular arithmetic of Cryptol in play here. Since only 3 bits are required to represent the numbers involved, all arithmetic is done modulo 2^3 , hence yielding $3 + 6 = 1$.) Nested records can be matched using a nested record pattern:

```
addDiff {p1 = {x = X; y = Y}; p2 = {x = X'; y = Y'}}
  = {p1 = {x = X+X'; y = Y+Y'}; p2 = {x = X-X'; y = Y-Y'}};
```

The inferred type for `addDiff` is:

```
addDiff : {a b c d} {p1 : {x : [a]b; y : [c]d};
p2 : {x : [a]b; y : [c]d}}
-> {p1 : {x : [a]b; y : [c]d};
p2 : {x : [a]b; y : [c]d}}
```

Here is a simple example use:

```
Cryptol> :set base=10
Cryptol> addDiff
{p1 = {x = 10; y = 20}; p2 = {x = 5; y = 10}} {p1={x=15; y=30};
p2={x=5; y=10}}
```

2.10.5 Field polymorphism

We expect that records will mostly be used for packing together various functions representing crypto-algorithms; i.e., as a means for specifying interface definitions. To fulfill this role, we will allow record fields to contain polymorphic fields, and in particular, polymorphic functions. (Note that the polymorphism will mostly come in the form of size-independence, where a given set of encryption/decryption functions can work over different key/block sizes.)

To keep the Cryptol record system simple, we require users to provide type signatures when polymorphic fields are used. We will also provide record patterns, which will allow type inference for records, but this inference will only provide monomorphic types unless the user provides an explicit type signature.

Here is a demonstration of a function making use of a polymorphic record field:

```
test : {b} {enc : {a} (a >= 2) => [a] -> b} -> [2]b;
test alg = [(alg.enc t1) (alg.enc t2)] where { t1 : [32]; t1 = 0;
t2 : [64]; t2 = 0
};
```

The idea is that the record argument passed to `test` contains an encryption function named `enc`, that can accept any argument that is larger than 2 bits wide. Also, `enc`'s output is polymorphic as well, i.e., `test` does not make any assumptions on what `enc` returns.

Here are some example uses of `test` as defined above:

```
Cryptol> test {enc x = x} : [2] [32]
[0x00000000 0x00000000]
      : [2] [32]
Cryptol> test {enc x = x} : [2] [16]
[0x0000 0x0000]
      : [2] [16]
```

Note the use of the polymorphic nature of the function `enc`.

When a field is used polymorphically, the user is required to provide a type signature. Type signatures can be omitted if record patterns are used (see Section 2.10.4, "Record patterns," on page 18), but in that case the inferred types are monomorphic. The following example demonstrates:

```
poly : {id : {a} a -> a} -> ([2], Bit);
poly r = (r.id 2, r.id True);

Cryptol> poly {id x = x}
(0x2, True)
```

The corresponding definition with a record pattern would fail to type check, since `id` is being used at two incompatible types:

```
mono {id} = (id 2, id True); // ill-typed!

Loading "tutorial.cry".. Checking types.
["tutorial.cry", line 10, col 17]: shape mismatch between [$24]
and Bit
```

Our advice is to use record patterns exclusively and rely on type inference. Only when a particular use-case explicitly requires the use of a polymorphic field, the users should annotate their types accordingly to tell Cryptol about the specific need.

2.10.6 Recursion

A record field can be defined recursively, both in terms of itself, or using the other fields that are being concurrently defined. Here is a simple example:

```
Cryptol> {xs = [1]#ys; ys = [2]#xs}.xs @@ [0 .. 10]
[0x1 0x2 0x1 0x2 0x1 0x2 0x1 0x2 0x1 0x2 0x1 0x2]
```

2.11 Type synonyms

One gripe about Cryptol is that the types can get overly complicated. Addition of record types to the type system can worsen the situation.

To overcome this problem, we allow type-synonyms, possibly parameterized by other types. Consider, for instance, the following type synonym declaration:

```
type Algorithm (keySize, blockSize) = {
  enc : ([keySize], [blockSize]) -> [blockSize];
  dec : ([keySize], [blockSize]) -> [blockSize];
};
```

Given this declaration, the test example from Section 2.10.4 can be given the following type:

```
test : {k b} (fin b) => (Algorithm (k, b), k, b) -> Bit
```

If type inference is used, the compiler would always infer types in their anonymous forms; without trying to match them to any declared synonyms in the scope. That is, given the above type declaration, we do not expect Cryptol to infer the type signature given above. Instead, if the user specifically writes this type down, Cryptol will make sure that the inferred type matches it.

Type synonyms are not just tied to records. Any old type can be shortened using a synonym appropriately. However, we anticipate that the typical use case are with records. The are described in detail in Section 3.13 on page 36.

3

Cryptol Language Elements

3.1 Comments

There are two forms of comments in Cryptol: single-line and multi-line, both following the forms and conventions of the C/C++/Java family of languages. A single-line comment starts with `//` and extends to the end of the line. Multi-line comments are enclosed by `/*` and `*/`. Multi-line comments can be nested.

EXAMPLE 3-1 *Using Comments*

```
/*  
Polymorphic bit twiddle /* tricky example */  
*/  
f x = x & - x; // Trick
```

3.2 Literate Cryptol

Literate programming is a coding style that emphasizes documentation. A literate Cryptol program, as opposed to a normal Cryptol program, is basically a text file where code sections are clearly marked. Any unmarked text is assumed to be a comment. (Compare this to the usual style of programming, where comments must be marked explicitly, and everything else is considered program text.)

This emphasis on documentation is especially useful when the same file is used for multiple purposes. For instance, the same literate Cryptol file written with LaTeX markup can be used to produce a nice document, while remaining a perfectly valid program to be read by the Cryptol interpreter. (In fact, the current document was prepared using the literate Cryptol style, to make sure that the very feature it is describing works as described!)

3.2.1 Bird tracks

One way to mark up code segments in a literate Cryptol program is to use the so called *bird tracks*:

This is a good old comment.

```
> foo = 1;  
> bar xs = [0] # xs;
```

Another comment here.

In this style, code blocks are marked by a singlet `>` character at the beginning of a line. Also, each bird-tracked code segment should be preceded and followed by at least one blank line. It is an error to have bird tracks and code segments to strictly follow each other; they should be separated by at least one blank line.

3.2.2 LaTeX-style markups

An alternate style is to use the markers `\begin{code}` and `\end{code}` to surround code blocks. (Clearly, this style is most useful when the Cryptol code is embedded in a LaTeX document, but the file can contain arbitrary text, which is simply going to be treated as comments.) Here is an example*:

```
\begin{code}  
xs = [foo] # bar ys;  
ys = [foo] # bar xs;  
\end{code}
```

In this style, there is no requirement to have a blank line preceding/following the marked-up code block. Comments can follow immediately before or after the code. Nesting of code blocks is not allowed and will be reported as an error to the user.

*. For use within LaTeX files, we found the following two environment definitions to be quite convenient:

```
\newenvironment{code}{\verbatim}{\endverbatim}  
\newenvironment{pseudoCode}{\verbatim}{\endverbatim}
```

The `pseudoCode` environment is most useful for including Cryptol text that is known to have errors in it. Of course, discriminating users can use fancier environment definitions that can produce line numbers, boxes, etc., as well.

3.2.3 Mix and match

The alert reader will have noticed that the previous examples also demonstrate how we can mix and match both literate styles in the same file. (In the second example above, we referred to the variables `foo` and `bar`, which were defined in the first example using bird tracks.)

The same is true for files included via Cryptol's `include` directive. We can include normal or literate Cryptol files using mixed style, and those files in turn can include both styles of files at will.

3.2.4 Valid file extensions

Cryptol programs are typically put in files that end with the `.cry` extension. With the addition of literate programming, it makes sense to use other suffixes as well. Here are the supported defaults:

- ◆ `.cry .cyl`: Normal Cryptol program,
- ◆ `.lcry .lcyl .tex`: Literate Cryptol program.

As usual, given a file name `f`, Cryptol interpreter will try to locate the file using all these possible extensions. The order of search will be: `f`, `f.cry`, `f.cyl`, `f.lcry`, `f.lcyl`, and finally `f.tex`; where the first match will be taken.

If there is no extension on a file name, or if the extension is not one of `.lcry`, `.lcyl`, or `.tex`, then the Cryptol program will assume it is a non-literate Cryptol program. (Consider, for instance, `"a.b.txt"`, which will be treated as a non-literate program.)

It is illegal to use literate programming mark-ups in a file that is considered non-literate. (That is, such mark-ups will be considered part of program text, which will most likely cause parsing errors.)

Ordinary Cryptol comments (i.e., `//` or `/* .. */` blocks) can be used as usual in literate file code segments.

3.2.5 Summary

Literate programming style is especially convenient for proper documentation of Cryptol programs, making sure that the Cryptol code that is placed in documents are free of syntax/type errors, as one can simply process the document using the Cryptol interpreter. We encourage programmers to use the literate style especially when writing Cryptol programs that are mainly intended to be read by other people (as in tutorials, papers, etc.).

3.3 Arithmetic

The basic arithmetic operators are defined over words. Arithmetic in Cryptol is modulo the size of the word. Thus, a 9-bit word will use arithmetic modulo 2^9 . The operators are $+$, $-$, $*$, $/$, $\%$ (modulo), and $**$ (power). When an arithmetic operator is used with sequences, the operation will be performed elementwise on the words.

EXAMPLE 3-2 *Element-wise arithmetic*

```
Cryptol> [1 2 3 4] + [4 3 2 1]  
[5 5 5 5]
```

3.4 Polynomial Arithmetic

In addition to modular arithmetic, Cryptol also provides support for polynomial arithmetic over polynomials with binary coefficients. The specific operations supported are: multiplication, division and modulus over polynomials.

Polynomials constants are written using `<| >` as brackets around a polynomial term over the variable `x` with `^` used to denote exponents. For example, the polynomial $x^8 + x^5 + x + 1$ is written as:

```
<| x^8 + x^5 + x + 1 |>
```

The terms of the polynomial may be written in any order. The following denotes the same polynomial.

```
<| 1 + x^8 + x + x^5 |>
```

Polynomials constants are really just syntactic sugar for a sequence of bits. The elements of the sequence are the coefficients. In this encoding, the index of a coefficient in the sequence is the same as its exponent. The size of the sequence is the same as the highest numbered exponent plus one. The above polynomial is equivalent to the following word:

```
0x123 : [9]
```

The arithmetic operations over polynomials are called `pmult`, `pdiv` and `pmod`, for polynomial multiplication, division and modulus respectively. Polynomial addition is just exclusive-or, and thus has no special operator.

3.5 Boolean operations

Cryptol has the standard Boolean operations: `&` (and), `|` (or), `^` (exclusive-or), and `~` (complement). They work on everything from bits to arbitrarily nested sequences. The operations on bits are standard, and on sequences they are performed elementwise.

3.6 Equality and Comparisons

The equality operator in Cryptol is written `==`. It compares two like-typed values, and returns a result of type `Bit`. Sequences are compared elementwise, to arbitrary depth. The operator `!=` checks for non-equality.

EXAMPLE 3-3 *Equality tests*

```
Cryptol> 1==2
False
Cryptol> [ [13 34] [14 9] ] == [ [13 34] [14 (3+6)] ]
True
Cryptol> [1 3 .. 10] == [1 3 5 7 9]
True
```

The standard comparison operators are available: `<`, `>`, `<=`, `>=`, and `!=`. These are only defined over words, and have a result of type `Bit`.

3.7 Conditional Expressions

The standard if-then-else construct is available. The first expression must be of type Bit, and the then and else expressions must be of the same type.

EXAMPLE 3-4 *Using if-then-else*

```
min (x, y) = if x <= y then x else y;
```

3.8 Shifts and Rotates

The shift and rotate operators in Cryptol allow shifting/rotating at the outer level of any sequence. The left-shift operator is `<<`, while `>>` shifts to the right. The sequence is filled in with zeros, where zero is understood to be an appropriately typed element that is all zeros. Left and right rotations are specified using the operators `<<<` and `>>>` respectively. For shifts and rotations, the left-hand argument is the sequence to be shifted/rotated, and the right-hand argument is a word describing how much to shift/rotate it.

EXAMPLE 3-5 Shifts and rotates

Assume we are in the context of the definitions:

```
xs = [0 1 2 3 4 5 6 7];
yss = [[0 1] [2 3] [4 5] [6 7]];
x = 0x8765;
```

Then we have:

```
Cryptol> xs << 4
[4 5 6 7 0 0 0 0]
Cryptol> xs <<< 4
[4 5 6 7 0 1 2 3]
Cryptol> yss >> 2
[[0 0] [0 0] [0 1] [2 3]]
Cryptol> x >> 8
0x0087
Cryptol> x >>> 4
0x5876
Cryptol> xs >> 10
[0 0 0 0 0 0 0 0]
Cryptol> xs >>> 8
[0 1 2 3 4 5 6 7]
Cryptol> xs >>> 10
[6 7 0 1 2 3 4 5]
```

Notice the wraparound effect for rotates, when the amount to rotate is larger than the size of the sequence.

If you have tried the above example in the Cryptol interpreter, you would have seen that all numbers are printed in hexadecimal. Normally Cryptol displays literals in hexadecimal, rather than decimal. However, it is possible to set the base to anything between 2 and 36 (inclusive). For instance, to set the output base to 13, use:

```
Cryptol> :set base=13
```

The typing of shift and rotate operators are discussed in Section 5.4, "Operators for Shifting and Rotating," on page 63.

3.9 Appending Sequences

Sequences of differing widths (but the same element types) may be appended together into one sequence using the append operator, #. For instance:

```
crypto1> [0 1 2 3] # [4 5 6]
[0 1 2 3 4 5 6]
```

3.10 Joining and Splitting Sequences

A common step in crypto-algorithms is to divide words into smaller pieces, such as dividing a 128-bit word into 4 32-bit words or joining small pieces together. Cryptol provides two built-in functions to make this particularly convenient. Consider the following function:

$$\text{frotz } [a \ b \ c \ d] = a + b - c + d;$$

We can pass a sequence consisting of four words to it, but if we want to call it with a 32-bit word we have to split the 32-bit word using the built-in function `split`. If we wanted to pass a sequence of type `[2][2][8]` we would have to join the outer level using the built-in function `join`. If we assume that we're doing 8-bit arithmetic (i.e., `a b c d` are all 8 bits wide), then we can either pass `frotz` a value of type `[4][8]`, `[32]`, or `[2][2][8]` by using the appropriate `split/join` function. For instance, assuming `frotz` is defined as above, we have:

EXAMPLE 3-6 *Using join and split*

```
Cryptol> frotz [5 6 7 8]
12
Cryptol> frotz (join [[5 6] [7 8]])
12
Cryptol> frotz (split 0x5678)
14
```

The reader may be surprised by the last example above. But recall that literals are little-endian; hence, when `frotz` is called, we have `a=8`, `b=7`, `c=6`, `d=5`, leading to the result 14.

3.11 Sequence Comprehensions

A very convenient way of describing sequences is the sequence comprehension notation, analogous to set comprehension notation. A sequence is described by drawing and combining elements from other sequences, referred to as generators. When there are multiple generators, the elements produced form the cartesian product of the two sequences.

EXAMPLE 3-7 *Simple sequence comprehensions*

```
crypto1> [| [x y] || x <- [0 1], y <- [4 .. 7] |]  
[[0 4] [0 5] [0 6] [0 7] [1 4] [1 5] [1 6] [1 7]]  
  
crypto1> [| [x y z] || x <- [0 1], y <- [4 .. 7]  
                  || z <- [0 .. 4]  
                  |]  
[[0 4 0] [0 5 1] [0 6 2] [0 7 3] [1 4 4]]
```

As seen in the second example above, generators may also be combined in parallel. In this case, the elements produced will only be as long as the elements of the shortest group.

Note: When you actually type this expression in the Cryptol interpreter, you'll have to enter it all in one line. We have used multiple lines above for clarity. In actual program text, however, definitions can span multiple lines.

Parallel generators are very useful for doing elementwise operations between sequences. For example, if we didn't have exclusive-or built in, we could have defined it as follows:

```
xor (xs, ys) = [| (x & ~y) | (~x & y)  
                || x <- xs  
                || y <- ys  
                |];
```

3.12 Streams and Recursively Defined Sequences

Cryptol can also express sequences of unbounded size, which we'll call *streams*. Streams allow us to model such things as shift registers, recurrence relations and cryptographic modes. The simplest way to define a stream is by using an unbounded sequence enumeration:

```
[0 ..]
```

which stands for the infinite stream of natural numbers. Streams can also be defined recursively. We can model a simple counter with the following definition:

```
counter init = [init] # counter (init + 1);
```

The streams `[0..]` and `counter 0` denote the same thing.

The size of a stream is indicated in the type system by a special constant named `inf`. In the above example, `counter` has type:

```
{a} (a >= 1) => [a] -> [inf][a]
```

The constraint `(a >= 1)` arises from addition by 1 in the definition of `counter`.

3.13 Type Synonyms

Types in Cryptol can get fairly complicated, especially in the presence of records. Even for simple types, it is desirable that mnemonic names can be used for readability and documentation purposes. Type synonyms facility addresses this problem by allowing users to give names to arbitrary types. In this sense, they are akin to typedef declarations in C. However, Cryptol's type synonyms are significantly more powerful because they can be parameterized by other types.

The extension of Cryptol with type synonyms is designed to be backwards compatible with existing Cryptol programs. The only caveat is that the name `type` is now a keyword. Therefore, existing Cryptol programs using this particular identifier will have to rename it accordingly.

In this section, we go over a number of examples to illustrate type synonyms.

3.13.1 Non-parameterized type synonyms

Without parameters, type synonyms act similar to C's typedef declarations. Consider, for instance:

```
type Bool = Bit;
```

Once this declaration is made, the names `Bit` and `Bool` can be used interchangeably. For instance, the above declaration would allow us to write:

```
and : (Bool, Bool) -> Bool;  
and (x, y) = x & y;
```

which is arguably more readable.

3.13.2 Parameterized synonyms

Type synonyms become more interesting when they are parameterized:

```
type EllipticCurvePoint aw  
= { isZero : Bit; x : [aw]; y : [aw] }; \end{code}
```

The above example has a single parameter, but we can have an arbitrary number of parameters as well. In this case the parameter names must be put into a tuple:

```
type Message mw = [mw];  
  
type MessagePackage (nm, ms1, ms2)  
= { m120 : [nm] (Message ms1);  
    m260 : [nm] (Message ms2)  
  };  
  
type KeyPair (aw)  
= { private : [aw];  
    public : EllipticCurvePoint aw  
  };  
  
type TestPackage (nm, nk, ks, ms1, ms2)  
= { msgpack : MessagePackage (nm, ms1, ms2);  
    keypack : [nk] (KeyPair ks)  
  };
```

When there is only one parameter, the parentheses are optional, as demonstrated by `Message` and `KeyPair` examples above. Also note that type synonyms themselves can refer to other type synonyms in their definitions. For instance, the definition of `TestPackage` uses the type synonym `KeyPair`, which in turn uses the synonym `EllipticCurvePoint`.

3.13.3 Type constants

An interesting application of type synonyms is to name type constants:

```
type size = 4;

foo : [size] -> [size];
foo x = x;
```

It is OK to use synonyms in type annotations as well:

```
baz : [size] -> [size];
baz x = x + (zero:[size]);
```

However, such constants are not visible in expression contexts. The following use will be rejected:

```
bar : [size] -> [size];
bar x = x + size; // illegal! will be rejected.
```

3.13.4 Nesting

As we hinted before, type synonyms can refer to other type synonyms as appropriate. Here is a simple example:

```
type Q a = S (a, a);
type S (a, b) = R b;
type R a = a;

z : Q Bit;
z = True;
```

3.13.5 Use in predicates

Type synonyms are also visible in predicates. Here's a contrived, but hopefully enlightening example:

```
type Id a = a;

f : {a} (Id a >= 2) => Id a;
f = undefined;
```

The scoping rules with type schemes apply as usual. For instance, in the following definition for `g`, the synonym name `same` is hidden by the scheme, but `pair` is visible as expected:

```
type Single a = a;
type Pair a = (a, a);

// The `Single` below is just a name, not the synonym above!

g : {Single} Single -> Pair Single;
g x = (x, x);
```

Indeed, if we ask Cryptol what the type of `g` is, it'll respond with:

```
Cryptol> :t g
g : {Same} Same -> (Same, Same)
```

Note that the synonym `Pair` is completely expanded in the resulting type expression.

To avoid confusion, we recommend using identifiers starting with lower-case letters for type variables, and identifiers starting with upper-case letters for type synonyms. While this convention is not enforced by Cryptol, it is a good idea to stick to it for readability and maintenance purposes.

Recursive synonyms

Cyclic dependencies (either direct or indirect) are not allowed, and they will be caught by Cryptol. Consider the following declaration:

```
type C a = C a;
```

Cryptol will respond with:

```
..location..} Type synonym "C" is cyclically defined
```

Indirect cycles are caught similarly:

```
type Foo a = Bar (a, a);
type Bar (a, b) = Baz (a, b, b);
type Baz (a, b, c) = Foo a;
```

We will get:

```
..location..} Cyclic dependency detected between type
synonyms (Foo, Bar, Baz)
```

3.13.6 Parameter saturation

All the type variables used in the definition should be mentioned in the parameters list:

```
type Bad a = (a, b);
```

This definition will cause Cryptol to complain: \

```
..location..} Undefined type "b"
```

However, beware of the potential confusion between type variables and synonym names. The following will be accepted fine:

```
type StillBadButOK a = (a, b);
type b = Bit;
```

since the occurrence of `b` will be taken to refer to the definition below, i.e., as a synonym for `Bit`.

Such pitfalls are best avoided by the coding style we mentioned before: Use uncapitalized identifiers for type variables, and capitalized identifiers for synonym names. The type synonym definition for `b` in the last example above violates this style, causing the confusion.

It goes without saying that correct number of arguments should be used. Both of the following examples will be rejected by Cryptol:

```
type B a = (a, a);

f : B;
f = undefined;

g : B (Bit, Bool);
g = undefined;
```

The error messages are (respectively):

```
..location..} Type synonym "B" requires 1 arguments, but
               is given none.
..location..} Type synonym "B" requires 1 arguments, but
               is given 2.
```

3.13.7 Phantom types

While using an unnamed parameter is an error, having parameters that are not actually used is perfectly OK:

```
type Line (n, a) = [n][8];
```

Note that the type variable `a` is not used in the definition.

Phantom types are mainly useful for documentation purposes:

```
sumLines : {n a} (Line (n, a), Line (n, a)) -> Line (n, a);
sumLines (l1, l2) = l1 + l2;
```

Intuitively, the type `Line (n, a)` represents sequences of length `n` that contain type `a` objects. The implementation however, represents all objects by 8 bit words. However, the type of `sumLines` is still useful since it states that it is intended to be used to sum lines that carry the same object type.

It is important to emphasize that phantom types are only for documentation purposes. To illustrate, let us define:

```
type Apples = [8];
type Oranges = [8];
type Pears = [8];
```

Using these synonyms, we can still add apples to oranges and get pears:

```
addApplesToOranges :
  {n} (Line (n, Apples), Line (n, Oranges))
  -> Line (n, Pears);
addApplesToOranges (as, os) = sumLines (as, os);
```

The function `addApplesToOranges` will type-check fine, since after expansion the involved types match precisely. Still, phantom types can be quite useful in expressing type “conventions” maintained by the programmer.

3.13.8 Simple kinds

All the type variables in a type-synonym declaration has to correspond to a concrete type. In particular, they cannot be applied to other type parameters. The following example illustrates:

```
type H a = a Bit;
```

This declaration will be rejected by Cryptol:

```
..location..} Unsupported higher-order type parameter "a"  
              in type synonym declaration for "H" }
```

Such declaration do make sense in languages with higher-order data types, since any use of them would require a partial application. (Think of how you might ever use the synonym `H` above; it would require an argument that is itself a type constructor that is partially applied.) Since Cryptol does not have such data types, we reject the corresponding type synonym declarations as well.

3.13.9 Practical implications

Type synonyms are only a syntactic convenience. In particular, they are completely “translated away” early in the compilation pipeline; that is, the Cryptol interpreter does not keep track of defined type synonyms after they are fully expanded. In theory, this is of little consequence. In practice, however, types inferred by the compiler (or those used in error messages) will have no knowledge of the available type synonyms. Therefore, such messages will always refer to types in their anonymous, fully expanded forms.

3.14 Pattern Bindings

Cryptol bindings mainly come in two flavors: bindings of simple variables or of functions.

```
x = 1;
f (x, y) = (x+y, x-y);
```

Such bindings can appear both at the top level, or within where clauses. Pattern bindings extend this notation so that the left hand side can be an arbitrary pattern. For instance:

```
(fst, snd) = (1, 2);
[a b c d e] = [1 2 3 4 5];
{fa; fb = (fb1, fb2)} = {fa = 1; fb = (2, 3)};
```

As expected, these bindings result in the following assignments: `fst=1`, `snd=2`, `a=1`, `b=2`, `c=3`, `d=4`, `e=5`, `fa=1`, `fb1=2`, and `fb2=3`. Note that the binding patterns can be arbitrarily nested, and they can appear both at the top-level and within where clauses.

Pattern bindings are most useful when extracting the results of a function call. Using the function `f` defined above, we can state:

```
(sum, diff) = f (10, 2);
```

Without pattern bindings, the above binding would have to be written:

```
sum_diff_temp = f (10, 2);
sum = project (1, 2, sum_diff_temp);
diff = project (2, 2, sum_diff_temp);
```

The reader can appreciate how much coding pattern bindings would save (especially in the presence of records), since all the extraction functionality would have to be written by the programmer otherwise. Also, pattern bindings saves the programmer from using temporary variables to hold intermediate results, such as `sum_diff_temp` above, avoiding name-space pollution.

Pattern bindings are implemented fairly early in the compilation pipe-line, basically getting translated as in the `sum/diff` example above. Usual rules about variable duplication still applies:

```
(z, (y, z)) = (2, (3, 4));
```

This definition would generate an error stating `z` is multiply defined.

Note that pattern bindings can define variables recursively:

```
(xs, ys) = ([0]#ys, [1]#xs);
```

This binding defines `xs = [0 1 0 1 ...]` and `ys = [1 0 1 0 ...]` where `xs` and `ys` are both infinite streams.

Clearly, a pattern binding has to respect the types. Consider:

```
(p1, p2) = True;
```

This binding will cause the following type error:

```
shape mismatch between ($54,$55)
and Bit
```

However, it is possible to have a run-time failure with sequences. Consider:

`[l1 l2] = [1];`

This definition will be accepted. When we query the value of `l2`, we get:

`index out of bounds (1 out of 1)`

(Recall that sequence indexing starts from 0.)

4

An Extended Example

This chapter provides a more detailed introduction to the Cryptol language by providing a walkthrough of a specification of the Advanced Encryption Standard. This specification of the AES algorithm follows the definition given in the AES Proposal document titled “The Rijndael Block Cipher.”

4.1 Preliminaries

The AES algorithm is an iterated block cipher with a block length of 128 and a variable key length, defined to be one of 128, 192 or 256 bits. As per the AES specification, the definition of the algorithm is parameterized by two symbolic constants, N_b and N_k . These two parameters define the number of 32-bit words per block and number of 32-bit words per key respectively. The block size is fixed at 128, and we will set the key size to be 128 for the purposes of this example. We will start by giving definitions to these constants:

```
Nb = 128 / 32;    // number of 32- bit words per block
Nk = 128 / 32;    // number of 32- bit words per key
```

The number of rounds that the cipher is iterated is also defined as a symbolic constant, but is derived from the values of N_b and N_k . The following definition is derived from the table given in the AES definition:

```
Nr = max(Nb, Nk) + 6;
```

Internally within the algorithm, intermediate cipher values are structured as a two-dimensional array of bytes, called the *State*, which has the following type:

```
[4][Nb][8]
```

That is, four blocks of N_b -many eight-bit words (bytes).

One of the first jobs of the cipher is to transform the input data, which we'll take as an array of bytes, into the internal form of the *State*, as you'll see in the definition of the top-level encrypt function:

```
encrypt (RK, PT) = unstripe (Rounds (State, RK))
  where {
    State : [4][Nb][8];
    State = stripe PT;
  };
```

This defines a function named `encrypt`, which takes two arguments: the round keys, called `RK`, and the plaintext to be encrypted, called `PT`. The result of the function is given by the expression after the equals sign. The flow of data in this definition can be found by tracing the dependencies between values. We see that the result of this function is expressed as a call to the function `unstripe`, which itself is applied to the result of calling the `Rounds` function. The `where` clause provides local definitions. In it, we define a new value that is the input plaintext transformed into the form of the *State* using the function `stripe`. Thus we see that the flow of data is as follows:

1. transform the input plaintext into the *State* using the `stripe` function
1. apply the `Rounds` function on the *State* and round keys to yield the final *State*
2. transform this final state into an array of bytes using `unstripe`.

The conversion function `stripe` is defined as follows:

```
stripe : [4*Nb][8] -> [4][Nb][8];
stripe block = transpose (split block);
```

This converts a one-dimensional array of bytes into the two-dimensional array of the State. The most straightforward way to do this is to use the built-in function `split`, as you see above. However, `split` lays out the data in a row-major fashion, while the specification calls for a column-major layout. Hence, after the `split`, the result is transposed to get column-major layout.

The corresponding unstriping function does the inverse:

```
unstripe : [4][Nb][8] -> [4*Nb][8];
unstripe state = join (transpose state);
```

The round keys have to be constructed from an initial key. We define a key schedule function to do this. It takes the key (an array of bytes) and expands it into a sequence of round keys, each of which has the same shape as the State. There is one for each round of the cipher (including the final round), plus one additional for the prelude. Thus, the key schedule produces a sequence of $Nr+1$ State values. Since the prelude and the final round are separate from the main iteration, it is convenient to keep their corresponding pieces of key material distinct; thus we can think of the key schedule phase as producing three values: key material for the prelude, the round keys for the inner rounds, and key material for the final round. This is expressed using a tuple of three values, which has the following type:

```
(([4][Nb][8], [Nr- 1][4][Nb][8], [4][Nb][8])
```

Thus the type for the key schedule function will be:

```
keySchedule : [4*Nk][8] -> ([4][Nb][8], [Nr- 1][4][Nb][8], [4][Nb][8]);
```

Now we can define the type of the encrypt function above. It takes a triple representing the key schedule, together with the plain text, and produces the cipher text.

```
encrypt : (([4][Nb][8],[Nr- 1][4][Nb][8],[4][Nb][8]), [4*Nb][8]) -> [4*Nb][8];
```

4.2 The Cipher Rounds

The Rounds function specifies the data flow within the cipher itself.

```
Rounds (State, (initialKey, rndKeys, finalKey)) = final
  where {
    istate = State ^ initialKey;
    rnds = [istate] # [| Round (state, key)
                  || state <- rnds
                  || key <- rndKeys |];
    final = FinalRound (last rnds, finalKey);
  };
```

This function uses the Cryptol idiom for specifying iteration: construct a stream of intermediate values, the first of which is the initial value, then one for each step of the iteration, and the last of which is the result of the iteration. The local value `rnds` is the example of this. The prelude for the AES cipher consists simply of exclusive-or'ing each byte of the incoming state with the initial key material. In the definition above, the result of the exclusive-or is called `istate`, because it is the initial state for the rounds iterations.

The value `rnds` is defined as a recursive sequence. The first element of the sequence is the value of `istate`. The second element is the result of applying the `Round` function on the first element of the `rnds` sequence (i.e. `istate`) and the first element of the `rndKeys` sequence. The third element of `rnds` is the result of applying the `Round` function on the second element of `rnds` (which is the result of the previous iteration) and the second element of `rndKeys`. This pattern continues until we exhaust `rndKeys`, at which point the sequence `rnds` is complete and its last element is the result of iterating the `Round` function once for each element of `rndKeys`.

4.3 The Round function

The Round function itself is composed of three different transformation steps followed by a mixing in of key material by an exclusive-or (as in the prelude).

```
Round : ([4][Nb][8], [4][Nb][8]) -> [4][Nb][8];
Round (State, RoundKey) = State3 ^ RoundKey
  where {
    State1 = ByteSub State;
    State2 = ShiftRow State1;
    State3 = MixColumn State2;
  };
```

Note that we've explicitly named the result of each transformation step. This is not necessary. We could have simply composed the function calls as follows:

```
Round (State, RoundKey) = MixColumn (ShiftRow (ByteSub State)) ^ RoundKey
```

However, in a Cryptol specification it is good style to name intermediate values that might be important. This is because when an intermediate value is named, it can be later accessed without making any modifications to the source. For example, the trace facility of the interpreter allows the user to trace the intermediate values of any named variable in a Cryptol specification.

4.3.1 The ByteSub Transformation

The ByteSub transformation specifies a non-linear byte-substitution, operating on each of the bytes of the State independently. In Cryptol, this is expressed as a nested sequence comprehension: first we iterate over each row of the state, then for each row, we iterate over each element of that row, applying the S-box transform to it.

```
ByteSub : [4][Nb][8] -> [4][Nb][8];
ByteSub state = [ [ Sbox x | x <- row ] | row <- state ];
```

The S-box transform is constructed by the composition of two transformations: first, taking the multiplicative inverse in $\text{GF}(2^8)$, and then applying an affine transformation over $\text{GF}(2)$. An S-box is typically pre-calculated and in-lined in an implementation for efficiency, since the tables are small. In Cryptol, the best approach is to specify tables from their high-level description (if available). Then, a table-based version can be easily derived. Although the table-based version is unnecessary from a specification point of view, it is often convenient from a validation point of view, as the Cryptol table can be directly compared to the implementation table. In addition, as a pragmatic concern, the table-based approach can speed up the execution of Cryptol specs in the interpreter considerably. We shall use a table-based approach here. Additionally, we shall hide the table behind a function call, so that person using the spec doesn't need to be aware that a table underlies it. The "trick" to doing this is quite straightforward, and takes advantage of the fact that a function of type $[8] \rightarrow [8]$ is isomorphic to a table of type $[256][8]$. First, define the table of all S-box values, then define a function that does the table lookup. The function `Sbox` is the one used in the rest of the specification.

```
sbox : [256][8];
sbox = [| affine (inverse x) || x <- [0 .. 255] |];

Sbox : [8] -> [8];
Sbox x = sbox @ x;
```

In order to find the inverse in $GF(2^8)$, we have to set up some basic Galois field arithmetic support. Fortunately, we can leverage of the existing support for polynomial arithmetic, since Galois field arithmetic can be represented using polynomials. In particular, multiplication in $GF(2^8)$ corresponds with multiplication of polynomials modulo an irreducible polynomial of degree 8. The choice of irreducible used in the definition of AES is $x^8 + x^4 + x^3 + x + 1$, which you'll see in Cryptol form below in the definition of `irred`. The polynomial built-ins `pmult` and `pmod` are then used to define multiplication and exponentiation in this Galois field (`gtimes`, `gpower`).

```
irred = <| x^8 + x^4 + x^3 + x + 1 |>;

gtimes : ([8], [8]) -> [8];
gtimes (x, y) = pmod (pmult (x, y), irred);

gpower (x, i) = ps @ i
  where ps = [1] # [| gtimes (x, p) || p <- ps |];
```

With the Galois arithmetic in place, we can then specify the inverse by a simple search, which relies on the Field property that each element has a unique inverse. Build a table of the number multiplied by every other number in the field, and find the unique index where the 1 is found.

```
inverse x = if x == 0 then 0 else find1 (ys, 0)
  where {
    ys = [| gtimes (x, y) || y <- [0 .. 255] |];
  };

find1 : ([256][8], [8]) -> [8];
find1 (xs, i)
  = if xs @ i == 1 then
    i
  else
    find1 (xs, i + 1);
```

The use of the table to specify the S-box highlights the distinction between what part of the specification describes the run-time part of the algorithm and the compile-time part of the algorithm that can be calculated ahead of time. The technique of using a search to find the arithmetic inverse would never be appropriate for the main algorithm, but is entirely appropriate in specifying configuration data that can be calculated ahead of time.

4.3.2 The ShiftRow Transformation

The ShiftRow transformation shifts the rows of the State cyclically. Each successive row is rotated one more position than the previous row.

```
ShiftRow : [4][Nb][8] -> [4][Nb][8];
ShiftRow state =
  [| row <<< i
```

```
|| row <- state
|| i <- [0 .. 3] ||;
```

4.3.3 The MixColumn Transformation

The MixColumn transformation considers the State as a sequence of columns, and each column is interpreted as a polynomial over $\text{GF}(2^8)$ and multiplied modulo $x^4 + 1$ with a fixed polynomial. As described in the specification paper, we can interpret the polynomial as a matrix and these operations as matrix multiplication. At the top-level of this transformation it is quite straightforward – we apply the multiplication operation with the fixed polynomial over each column. This is done by first transposing, so that the matrices are in column major order, applying the `multCol` transform once per column, then transposing back.

```
MixColumn : [4][Nb][8] -> [4][Nb][8];
MixColumn state =
  transpose [| multCol (cx, col)
             || col <- transpose state
             |];
```

The `multCol` function defines the multiply of a matrix by a vector (one column of the state).

```
multCol (cx, col) = join (mmult (cx, split col));
```

It defines the multiplication in terms of a standard matrix by matrix multiply function, and thus the vector must first be promoted to a matrix by using the following instance of `split` to make each vector element into a singleton vector:

```
split : [Nb][8] -> [Nb][1][8]
```

The `polyMat` function converts a representation of the polynomial as a sequence of coefficients into the matrix that is used for the matrix multiplication.

```
cx = polyMat [ 0x02 0x01 0x01 0x03 ];
polyMat coeff = transpose (take (width coeff, cols))
  where cols = [coeff] # [| cs >>> 1 || cs <- cols |];
```

4.4 The Key Schedule

The Key Schedule consists of two components: key expansion, and round key selection. We will first consider key expansion.

4.4.1 Key Expansion

The expanded key is a sequence of 4-byte words of length $(Nr+1)*Nb$. We will denote this with the Cryptol type $[(Nr+1)*Nb][4][8]$. In the following, we will use “word” to mean four bytes, since that is a more convenient representation than 32-bits. The first Nk elements of this sequence contain the key itself, with the remaining elements being defined recursively in terms of previous ones.

```
keyExpansion : [4*Nk][8] -> [(Nr+1)*Nb][4][8];
keyExpansion key = W
  where {
    keyCols : [Nk][4][8];
    keyCols = split key;
    W = keyCols # [| nextWord (i, old, prev)
                    || i <- [Nk .. ((Nr+1)*Nb-1)]
                    || old <- W
                    || prev <- drop (Nk-1, W)
                  |]; }
```

The `nextWord` function calculates the value for the next word in the expanded key sequence. Each word produced is calculated from the word Nk positions earlier in the sequence (`old`) and the previous word in the sequence (`prev`). Notice that `old` is defined as drawing from `W`. Since `keyCols` intervenes between where the sequence comprehension starts and the beginning of `W`, there will be Nk (the length of `keyCols`) elements between the value that `old` denotes and the current value in the comprehension. In order to get at the previous word in the sequence to define `prev`, since we know that there will be a difference of Nk elements between `W` and the start of the comprehension, we can get at the previous element by dropping $Nk-1$ elements off of the front of `W`.

In addition to exclusive-or'ing the `old` and `prev` words, for words in positions that are a multiple of Nk , a transformation is applied to `prev` prior to the exclusive-or. This transform consists of a rotation of the bytes in a word. The amount of the rotation is specified by AES to depend on the keysize, i.e. on the value Nk . The rotation is followed by the application of the `sbox` to the individual bytes of the word (`subByte`), then exclusive-or'ing in a round constant (`Rcon`). Since these operations are all byte-oriented, this is why the representation of a word as four bytes was chosen for the key expansion.

```
nextWord : ([8],[4][8],[4][8]) -> [4][8];
nextWord (i, old, prev) = old ^ prev'
  where prev' =
    if i % Nk == 0 then
      subByte (prev <<< 1) ^ Rcon (i / Nk)
    else if (Nk > 6) & (i % Nk == 4) then
      subByte prev
    else prev;

Rcon i = [(gpower (<| x |>, i - 1)) 0 0 0];

subByte p = [| Sbox x || x <- p |];
```

The “round constant” $Rcon$ is defined as a word whose first byte is the value $x^{(i-1)}$ in the field $GF(2^8)$ (the same field as used earlier in the S-box).

4.4.2 Round Key Selection

The final part of the key schedule is the round key selection, which we’ll bundle into the top-level `keySchedule` function. In this function, apply the key expansion, then re-arrange the expanded key into the form of the round keys, and finally split out the initial and final round keys from the rest of the round keys to make the 3-tuple used by the main cipher. The transformation into the round keys is accomplished by first applying `split` on the expanded key, yielding a sequence of type `[Nr+1][Nb][4][8]`, then a transpose is done on the elements of this sequence, yielding the final form, as seen by the type annotation on `rKeys`.

```
keySchedule : [4*Nk][8] -> ([4][Nb][8], [Nr-1][4][Nb][8], [4][Nb][8]);
keySchedule Key = (rKeys @ 0, rKeys @@ [1 .. (Nr-1)], rKeys @ Nr)
  where {
    W : [(Nr+1)*Nb][4][8];
    W = keyExpansion Key;
    rKeys : [Nr+1][4][Nb][8];
    rKeys = [| transpose ws || ws <- split W |];
  };
```

We can now put all the pieces together to define a top-level function that specifies AES encryption.

```
aes : ([4*Nk][8], [4*Nb][8]) -> [4*Nb][8];
aes (Key, PT) = encrypt (rkeys, PT)
  where rkeys = keySchedule Key;
```


5

Predefined Functions and Values in Cryptol

Cryptol comes with a number of predefined primitive functions and values, which can be collectively referred to as the Cryptol base library. This chapter takes a brief look at the functionality provided by this library.

5.1 Arithmetic Operators

This section reviews arithmetic operators in Cryptol.

5.1.1 + (Addition)

$+$: {a b} ([a]b,[a]b) -> [a]b

Over words, adds two words, modulo 2^a . Over higher-dimensional sequences, operates element-wise.

EXAMPLE 5-1 Using +

In the first example below, each literal can fit into 3 bits, and hence arithmetic is modulo $2^3 = 8$, but in the second, the presence of the literals 8 and 9 gives rise to the use of 4 bits; hence arithmetic is done modulo $2^4 = 16$. Notice that sequences are uniform, i.e. every element in a sequence has the same type.

```
Cryptol> [(4+0) (4+1) (4+2) (4+3) (4+4) (4+5) (4+6) (4+7)]  
[4 5 6 7 0 1 2 3]  
Cryptol> [(4+0) (4+1) (4+2) (4+3) (4+4) (4+5) (4+6) (4+7) (4+8) (4+9)]  
[4 5 6 7 8 9 10 11 12 13]
```

Addition extends to higher-dimensional sequences as well, as illustrated in this last example:

```
Cryptol> [[1 2] [3 4]] + [[2 3] [4 5]]  
[[3 5] [7 1]]
```

5.1.2 - (Subtraction)

$-$: {a b} ([a]b,[a]b) -> [a]b

Over words, subtracts two words, modulo 2^a . Over higher-dimensional sequences, operates element-wise. Subtraction is defined as:

$a - b = a + \text{negate } b$

EXAMPLE 5-2 Using -

Notice that there are no negative literals in Cryptol! In the following example, the use of literals 8 and 9 require 4 bits, and hence the arithmetic is done modulo $2^4 = 16$.

Notice how the results wrap around after 0.

```
Cryptol> [(4-0) (4-1) (4-2) (4-3) (4-4) (4-5) (4-6) (4-7) (4-8) (4-9)]  
[4 3 2 1 0 15 14 13 12 11]
```

See also Section 5.3.7, "negate (Two's complement)," on page 61).

5.1.3 * (Multiplication)

$*$: {a b} ([a]b,[a]b) -> [a]b

Over words, multiplies two words, modulo 2^a . Over higher-dimensioned sequences, operates element-wise.

EXAMPLE 5-3 Using *

In both of these examples, we need 3 bits; hence the multiplication is done modulo 8.

```
cryptol> 4*3
4
cryptol> [2 3] * [3 4]
[6 4]
```

5.1.4 / (Division)

$/$: {a b} ([a]b,[a]b) -> [a]b

Over words, divides two words, modulo 2^a . Over higher-dimensioned sequences, operates element-wise.

EXAMPLE 5-4 Using /

```
cryptol> 11/2
5
cryptol> 11/3
3
cryptol> 11/4
2
cryptol> 11/6
1
```

Notice that division by 0 terminates evaluation and returns an error.

```
cryptol> 11/0
divide by zero
```

5.1.5 % (Remainder)

$\%$: {a b} ([a]b,[a]b) -> [a]b

Over words, returns the remainder of dividing the two words. Over higher-dimensioned sequences, operates element-wise.

EXAMPLE 5-5 *Using %*

```
Crypto1> 11 % 2
1
Crypto1> 11 % 3
2
Crypto1> 11 % 4
3
Crypto1> [7 11] % [2 8]
[1 3]
```

Again, % 0 is fatal!

```
Crypto1> 11 % 0
Floating point exception
```

5.1.6 **** (Exponentiation)**

****** : {a b} ([a]b,[a]b) -> [a]b

Over words, takes the exponent of two words, modulo 2^a . Over higher-dimensioned sequences, operates element-wise. Be careful—due to its fast-growing nature, exponentiation is particularly prone to interacting poorly with defaulting.

EXAMPLE 5-6 *Using ***

Watch out for defaulting and the effect of type signatures in the following examples:

```
Crypto1> 2 ** 3
0
Crypto1> x where { x : [4]; x = 2 ** 3; }
8
Crypto1> x where { x : [4]; x = 2 ** 4; }
0
Crypto1> x where { x : [5]; x = 2 ** 4; }
16
```

5.1.7 **lg2 (Logarithm base 2)**

lg2 : {a b} (fin a) => [a]b -> [a]b

Takes the ceiling log, base 2, of a number. Over higher-dimensioned sequences, operates element-wise.

EXAMPLE 5-7 *Using lg2*

Notice that the following two expressions mean exactly the same thing:

```
cryptol> [| lg2 i || i <- [1 .. 10] |]  
[0 1 2 2 3 3 3 3 4 4]  
Cryptol> lg2 [1 .. 10]  
[0 1 2 2 3 3 3 3 4 4]
```

5.2 Polynomial Operators

This section reviews polynomial values and operators in Cryptol.

5.2.1 Polynomial Constants

Polynomial constants are written using `<| |>` as brackets around a polynomial term over the variable `x` with `^` used to denote exponents. For example, the polynomial $x^8 + x^5 + x + 1$ is written as:

```
<| x^8 + x^5 + x + 1 |>
```

The terms of the polynomial may be written in any order. The following denotes the same polynomial.

```
<| 1 + x^8 + x + x^5 |>
```

Polynomial constants are syntactic sugar for a sequence of bits. The elements of the sequence are the coefficients. The index of a coefficient in the sequence is the same as its exponent. The size of the sequence is the same as the highest numbered exponent plus one. The above polynomial is equivalent to the following word:

```
0x123 : [9]
```

5.2.2 Polynomial multiplication

```
pmult : {a b} (fin a, fin b, a >= 1, b >= 1) => ([a],[b]) -> [a+b-1]
```

Multiplies two polynomials. Note that the size of the result is additive in the size of the arguments, unlike the regular arithmetic multiply.

5.2.3 Polynomial division

```
pdiv : {a b} (fin a, fin b, a >= 1, b >= 1) => ([a],[b]) -> [a]
```

Divides two polynomials.

5.2.4 Polynomial modulus

```
pmod : {a b} (fin a, fin b, a >= 1) > ([a],[b+1]) -> [b]
```

Returns the modulus of two polynomials.

5.3 Logical Operators

This section reviews logical values and operators in Cryptol.

5.3.1 False

False : Bit

The constant False corresponds to the bit value 0.

EXAMPLE 5-8 Using False

Notice that False is *not* 0! Even the types of these two things are different. But a bit vector of all False is essentially 0. Here are some examples to illustrate:

```
Cryptol> False == 0
["command line", line 1, col 7]:
shape mismatch between Bit and [a]

Cryptol> [False] == 0
True
Cryptol> [False False] == 0
True
```

5.3.2 True

True : Bit

The constant True corresponds to the bit value 1.

EXAMPLE 5-9 Using True

Same comments as False apply in this case as well: True is *not* 1. Notice the little-endianness in the last example:

```
Cryptol> True == 1
["command line", line 1, col 6]:
shape mismatch between Bit and [a]
Cryptol> [True] == 1
True
Cryptol> [True False] == 1
True
Cryptol> [False True] == 1
False
```

5.3.3 & (Conjunction)

& : {a} (a,a) -> a

Logical ‘and’ over bits. Extends element-wise over higher-dimensioned sequences.

EXAMPLE 5-10 *Using &*

Several examples of & follow. Notice that & is quite polymorphic in its type, although both arguments must be of the same type!

```
Crypto1> 1 & 2
0
Crypto1> 2 & 3
2
Crypto1> True & False
False
Crypto1> [0xc 0x3] & [0x3 0xc]
[0 0]
Crypto1> True & 2
["command line", line 1, col 6]:
shape mismatch between Bit and [a]
```

5.3.4 | (Disjunction)

| : {a} (a,a) -> a

Logical ‘or’ over bits. Extends element-wise over higher-dimensioned sequences.

Notice that | is quite polymorphic in its type, although both arguments must be of the same type!

EXAMPLE 5-11 *Using |*

```
Crypto1> 2 & 3
2
Crypto1> 1 | 2
3
Crypto1> 2 | 0
2
Crypto1> True | False
True
Crypto1> [0xc 0x3] | [0x3 0xc]
[0xf 0xf]
```

5.3.5 ^ (Exclusive-or)

^ : {a} (a,a) -> a

Logical ‘exclusive-or’ over bits. Extends element-wise over higher-dimensioned sequences.

Notice that ^ is quite polymorphic in its type, although both arguments must be of the same type!

EXAMPLE 5-12 *Using ^*

```

Crypto1> 1 ^ 2
0x3
Crypto1> 2 ^ 3
0x1
Crypto1> True ^ False
True
Crypto1> True ^ True
False
Crypto1> [0xc 0x3] ^ [0x3 0xc]
[0xf 0xf]

```

5.3.6 ~ (Complement)

$\sim : \{a\} a \rightarrow a$

Logical complement over bits. Extends element-wise over higher-dimensioned sequences.

EXAMPLE 5-13 *Using ~*

```

Crypto1> ~ 1
0x0
Crypto1> ~ 2
0x1
Crypto1> ~ True
False
Crypto1> ~ [0xc 0x3]
[0x3 0xc]

```

Notice how the size information is important in using `~`. In the following set, the last example is the same as the first, since the literal 2 only needs 2 bits at the top level.

```

Crypto1> ~x where {x : [2]; x = 2;}
1
Crypto1> ~x where {x : [3]; x = 2;}
5
Crypto1> ~x where {x : [4]; x = 2;}
13
Crypto1> ~2
1

```

5.3.7 negate (Two's complement)

$\text{negate} : \{a\} b \ (a \geq 1) \Rightarrow [a]b \rightarrow [a]b$

Returns the two's complement of its argument. Over higher-dimension sequences, operates elementwise. Negate is defined as:

$\text{negate } a = \sim a + 1$

EXAMPLE 5-14 *Using negate*

As in \sim , the size information is important in *negate* as well. Notice that the invariant here is $a + \text{negate } a == 0$.

```
Crypto1> negate x where { x : [2]; x = 2; }  
2  
Crypto1> negate x where { x : [3]; x = 2; }  
6  
Crypto1> negate x where { x : [4]; x = 2; }  
14  
Crypto1> negate x where { x : [5]; x = 2; }  
30
```

See also Section 5.3.6, " \sim (Complement)," on page 61.

5.4 Operators for Shifting and Rotating

This section reviews shift and rotate operators in Cryptol. See Example 3-5, "Shifts and rotates," on page 31 for example of use.

5.4.1 << (Left shift)

$$<< : \{a\ b\ c\} (c \geq \lg 2(a)) \Rightarrow ([a]b, [c]) \rightarrow [a]b$$

The first argument is the sequence to shift; the second is the number of positions to shift by. The constraint on the size of the shift value ensures that the shift value is able to specify shifts over the entire length of the shifted value. That is, if the shifted value is of length 8, then the shift value must be long enough to allow a bit to be shifted from the right-most position to the left-most position (shift of 7) in a single operation. So the shift value must have at least 3 bits. For further explanation, see Section 5.4.6, "Typing of shift and rotate operators," on page 64.

5.4.2 >> (Right shift)

$$>> : \{a\ b\ c\} (c \geq \lg 2(a)) \Rightarrow ([a]b, [c]) \rightarrow [a]b$$

The first argument is the sequence to shift; the second is the number of positions to shift by. The constraint on the size of the shift value ensures that the shift value is able to specify shifts over the entire length of the shifted value. That is, if the shifted value is of length 8, then the shift value must be long enough to allow a bit to be shifted from the left-most position to the right-most position (shift of 7) in a single operation. So the shift value must have at least 3 bits. For further explanation, see *Typing of shift and rotate operators*.

5.4.3 <<< (Left rotate)

$$<<< : \{a\ b\ c\} (c \geq \lg 2(a)) \Rightarrow ([a]b, [c]) \rightarrow [a]b$$

The first argument is the sequence to shift; the second is the number of positions to rotate by. The constraint on the length of the rotate value ensures that all possible rotation values are available in a single rotate operation. For further explanation, see *Typing of shift and rotate operators*.

5.4.4 >>> (Right rotate)

$$>>> : \{a\ b\ c\} (c \geq \lg 2(a)) \Rightarrow ([a]b, [c]) \rightarrow [a]b$$

The first argument is the sequence to shift; the second is the number of positions to rotate by. The constraint on the length of the rotate value ensures that all possible rotation values are available in a single rotate operation. For further explanation, see *Typing of shift and rotate operators*.

5.4.5 Shifting and rotating words

Although it might not be immediately apparent, words are a special case with shifting and rotating. Consider the following:

```
Crypto1> 0x1234 >> 4
0x0123
```

This provides the expected answer. However, the expected answer, while following the usual interpretation of right shift as “divide by power-of-two”, actually corresponds to a *left* shift of the word considered as a sequence of bits. We can see this more clearly by doing a right shift on a literal sequence of bits, and using format to show us what’s really going on:

```
Crypto1> :print format ("%[]B", [False True False False] >> 1)
[ True False False False]
```

This is an unavoidable consequence of the little-endian interpretation of words: we have the choice of making a right-shift be a right-shift of the sequence of bits, which is consistent with other sequences in the language, or a right-shift of the *presentation* of the word, which is consistent with the usual interpretation of “right shift.”

5.4.6 Typing of shift and rotate operators

The careful reader might have noticed the predicate in the type of shift and rotate operators, which is of the form:

$$\{a \ b \ c\} \ (c \geq \lg 2(a)) \Rightarrow ([a]b, [c]) \rightarrow [a]b$$

Here, the type variable b denotes the size of the elements of the sequence. The type variable a stands for the size of the sequence itself, and c is the size of the shift/rotate offset. At first, the predicate seems unnecessary. This constraint states that the number that specifies the amount to shift/rotate has to be at least as wide as the number of bits required to represent the size of the sequence. That is, the following is ill-typed:

```
Crypto1> [1 2 3 4] >> x where { x : [1]; x = 1; }
["command line", line 1, col 11
In primitive "shift":
Inferred size of 1 is too small - expected to be >= 2
```

Since the input has 4 elements, we need at least 2 bits to describe the maximum shift from the left-most to the right-most position, which is a shift of 3. 1 bit is not enough to represent the value 3! Notice that the following is OK:

```
Crypto1> [1 2 3 4] >> 1
[0 1 2 3]
```

since 1 is defaulted to be 2 bits wide by the top level.

The reason for this constraint is the following. Consider:

```
Crypto1> [| [1 2 3 4] << i || i <- [0 ..] |] @@ [0 .. 3]
[[1 2 3 4] [2 3 4 0] [3 4 0 0] [4 0 0 0]]
```

If we didn't have the constraint $c \geq \lg_2(a)$, the type of the variable i would be constrained to be 1 bit only, and i would assume the values $[0\ 1\ 0\ 1\ 0\ 1\ \dots]$. Therefore the shifts would be either 0 or 1 place at a time, giving the unexpected result:

```
[[1 2 3 4] [2 3 4 0] [1 2 3 4] [2 3 4 0]]
```

The constraint $c \geq \lg_2(a)$ makes sure that i is at least 2 bits wide, and solves this problem nicely. The curious reader might want to try:

```
cryptol> [| [1 2 3 4] << i || i <- xs |] @@ [0 .. 3]
               where { xs : [inf][1]; xs = [0..]; }
["command line", line 1, col 14
In primitive "shift":
Inferred size of 1 is too small - expected to be >= 2
cryptol> [| [1 2 3 4] << i || i <- xs |] @@ [0 .. 3]
               where { xs : [inf][2]; xs = [0..]; }
[[1 2 3 4] [2 3 4 0] [3 4 0 0] [4 0 0 0]]
```

5.5 Operators for Sequence Manipulation

This section reviews various operators that work on sequences in Cryptol.

5.5.1 width

$\text{width} : \{a\ b\ c\} (c \geq \text{width}(a)) \Rightarrow [a]b \rightarrow [c]$

Returns the number of elements in the top level of the argument sequence. Thus, for a word, this returns the word size, in bits.

EXAMPLE 5-15 *Using width*

Here are some examples showing the use of *width*. Notice that the literal 0 and the empty sequence has zero width.

```
cryptol> width [1 2 3 4]
4
cryptol> width [[1 2 3 4]]
1
cryptol> width []
0
cryptol> width 12
4
cryptol> width 0
0
```

5.5.2 @ (Select)

$@ : \{a\ b\ c\} ([a]b, [c]) \rightarrow b$

Index operator. The first argument is a sequence. The second argument is the zero-based index of the element to select from the sequence.

EXAMPLE 5-16 *Using @*

Here are some examples showing the use of @.

```
cryptol> [1 2 3] @ 0
1
cryptol> [1 2 3] @ 3
"command line", line 1, col 9: index out of bounds (3 out of 3).
```

5.5.3 @@ (Permute)

$@@ : \{a\ b\ c\ d\} ([a]b, [c][d]) \rightarrow [c]b$

Bulk indexing, useful as a permutation operator. Defined as:

$xs\ @@\ is = [\mid xs\ @\ i \mid i <- is]$

EXAMPLE 5-17 *Using @@*

Here are some examples showing the use of @@.

```
Cryptol> [1 2 3] @@ [2 0 2]
[3 1 3]
Cryptol> [1 2 3] @@ [1 4 1]
[2 "command line", line 1, col 9: index out of bounds (4 out of 3)].
```

We also see laziness in action in the last example. The run-time error comes after we see the first element of the resulting sequence.

5.5.4 ! (Select from the end)

$! : \{a\ b\ c\} \text{ (fin } a) \Rightarrow ([a]b, [c]) \rightarrow b$

Index from the end of a sequence. The first argument is a sequence. The second argument is the zero-based index, starting at the end of the sequence and working towards the beginning, of the element to select from the sequence.

5.5.5 !! (Reverse Permute)

$!! : \{a\ b\ c\ d\} \text{ (fin } a) \Rightarrow ([a]b, [c][d]) \rightarrow [c]b$

Index in bulk from the end of a sequence. Analogous to @@.

5.5.6 # (Concatenate)

$\# : \{a\ b\ c\} \text{ (fin } a) \Rightarrow ([a]b, [c]b) \rightarrow [a+c]b$

Concatenate two sequences. Note that the left-hand sequence is restricted to be finite.

EXAMPLE 5-18 *Using #*

Here are some examples showing the use of #.

```
Cryptol> [1 2] # [3 4 5]
[1 2 3 4 5]
Cryptol> [[1 2]] # [[3 4] [5 6]]
[[1 2] [3 4] [5 6]]
Cryptol> 2 # 3
14
```

In the last example, notice that $2 == [\text{False True}]$, $3 == [\text{True True}]$ and $[\text{False True True True}] == 14$.

5.5.7 split

$\text{split} : \{a\ b\ c\} [a*b]c \rightarrow [a][b]c$

Split a sequence. As seen from the type, `split` is quite flexible. If the `-d` option is set, the following will be printed:

```
Cryptol> split [1 2 3 4]
In a top-level expression: with inferred type:
{a b c} [a][b][c]
encountered the following unresolved constraints:
c >= 3
a*b == 4
```

This is Cryptol's way of saying that it doesn't have enough information to determine what the result type should be. To resolve this, the user must specify the type of the answer:

```
Cryptol> split [1 2 3 4] : [2][2][3]
[[1 2] [3 4]]
Cryptol> split [1 2 3 4] : [1][4][3]
[[1 2 3 4]]
Cryptol> split [1 2 3 4] : [4][1][3]
[[1] [2] [3] [4]]
```

In general, the type of a `split` will be determined by the surrounding context.

See also Section 5.5.8, "join," on page 68, Section 5.5.9, "splitBy," on page 68, and Section 5.5.10, "groupBy," on page 69.

5.5.8 join

`join : {a b c} [a][b]c -> [a*b]c`

Concatenate the elements of a sequence. You can consider `join` as `append` over many elements.

EXAMPLE 5-19 Using `join`

```
Cryptol> join [[1 2] [2 3] [3 4]]
[1 2 2 3 3 4]
Cryptol> join [1 2 3]
57
```

We leave it to the reader to figure out why the final example comes out as 57.

See also Section 5.5.7, "split," on page 67, Section 5.5.9, "splitBy," on page 68, and Section 5.5.10, "groupBy," on page 69.

5.5.9 splitBy

`splitBy (n, _) : {a b} [n*a]b -> [n][a]b`

Split a sequence n ways. Sometimes we don't need the generality of `split` (such as when we know exactly how many pieces we want). In that case, we can use `splitBy` rather than `split`.

EXAMPLE 5-20 *Using splitBy*

```
Cryptol> splitBy (3, [0 .. 14])
[[0 1 2 3 4] [5 6 7 8 9] [10 11 12 13 14]]
Cryptol> splitBy (3, [0 2])
In a top-level expression: with inferred type:
{a} [3][a][2]
encountered the following unresolved constraint:
3*a == 2
```

The last example demonstrates the fact that the size of the input must be an even multiple of the split factor. Cryptol tells us that it can't find a size (a), such that 3 times that number is 2.

See also Section 5.5.7, "split," on page 67.

5.5.10 groupBy

```
groupBy (n, _) : {a b} [a*n]b -> [a][n]b
```

Split a sequence into n -wide pieces. The function `groupBy` provides a complementary functionality to `splitBy`. Rather than giving n pieces, we get pieces each of which is n wide.

EXAMPLE 5-21 *Using groupBy*

```
Cryptol> groupBy (3, [0 .. 14])
[[0 1 2] [3 4 5] [6 7 8] [9 10 11] [12 13 14]]
Cryptol> groupBy (3, [0 2])
In a top-level expression: with inferred type:
{a} [a][3][2]
encountered the following unresolved constraint:
3*a == 2
```

The last example demonstrates the even-multiple-size requirement, just as in the case for `splitBy`.

See also Section 5.5.7, "split," on page 67.

5.5.11 take

```
take (n, _) : {a b} (fin n, a >= 0) => [n+a]b -> [n]b
```

Construct a new sequence by taking the first n elements of the argument sequence.

EXAMPLE 5-22 *Using take*

```
Cryptol> take (4, [1 .. 8])

[1 2 3 4]

Cryptol> take (4, [1])
```

```
["command line", line 1, col 1  
In primitive "take"]:  
Inferred size of -3 is too small - expected to be >= 0
```

As seen, the input sequence should be at least as long as the number of elements to take. The error message states that the argument sequence is three elements short of satisfying the length requirements.

5.5.12 drop

$\text{drop}(n, _) : (\text{fin } n, n \geq 0) \Rightarrow [n+a]b \rightarrow [a]b$

Construct a new sequence by leaving out the first n elements of the argument sequence.

EXAMPLE 5-23 *Using drop*

```
Crypto1> drop (4, [1 .. 8])  
[5 6 7 8]  
Crypto1> drop (4, [1])  
["command line", line 1, col 1]:  
Inferred size of -3 is too small - expected to be >= 0
```

As seen, the input sequence should be at least as long as the number of elements to drop. The error message states that the resulting sequence is three elements short of satisfying the length requirements.

5.5.13 tail

$\text{tail} : \{a\} b \rightarrow [a+1]b \rightarrow [a]b$

Drop the first element of a sequence. Defined as:

$\text{tail } x = \text{drop } (1, x)$

EXAMPLE 5-24 *Using tail*

```

cryptol> tail 12
6
cryptol> tail [1 2 3]
[2 3]
cryptol> tail []
["command line", line 1, col 1]:
Inferred size of -1 is too small - expected to be >= 0

```

In the first example, `12 == [False False True True]`, and `6 = [False True True]`. As seen in the last example, the input sequence should not be empty.

See also Section 5.5.12, "drop," on page 70.

5.5.14 reverse

`reverse : {a b} (fin a) => [a]b -> [a]b`

Reverses the (top-level) elements of the sequence.

EXAMPLE 5-25 *Using reverse*

```

cryptol> reverse [1 2 3 4]
[4 3 2 1]
cryptol> reverse [[1 2] [3 4]]
[[3 4] [1 2]]
cryptol> reverse [] : [0][1]
[]
cryptol> reverse 12
3

```

In the second example, we see that only the top-level elements are reversed, not the inner ones. In the third example, we see that empty sequence can be reversed, but we need to specify the type explicitly at the top level, otherwise we will get a too-polymorphic type. We leave it to the reader to figure out why `reverse 12` is 3.

5.5.15 transpose

`transpose : {a b c} [a][b]c -> [b][a]c`

Transpose the top two levels of the nested sequence.

EXAMPLE 5-26 *Using transpose*

```

cryptol> transpose [[1 2 3] [4 5 6] [7 8 9]]
[[1 4 7] [2 5 8] [3 6 9]]

```

5.6 Comparison Operators

This section reviews various operators that perform comparisons in Cryptol.

5.6.1 == (Equals)

$\text{==} : \{a\} \text{ (fin } a) \Rightarrow (a,a) \rightarrow \text{Bit}$

Compares any two values of the same type for equality.

5.6.2 != (Not equals)

$\text{!=} : \{a\} \text{ (fin } a) \Rightarrow (a,a) \rightarrow \text{Bit}$

Compares any two values of the same type for inequality.

5.6.3 < (Less than)

$< : \{a\} \text{ (fin } a) \Rightarrow ([a],[a]) \rightarrow \text{Bit}$

Only works on words.

5.6.4 <= (Less than or Equal)

$<= : \{a\} \text{ (fin } a) \Rightarrow ([a],[a]) \rightarrow \text{Bit}$

Only works on words.

$>$ (Greater than)

$> : \{a\} \text{ (fin } a) \Rightarrow ([a],[a]) \rightarrow \text{Bit}$

Only works on words.

5.6.5 >= (Greater than or Equal)

$>= : \{a\} \text{ (fin } a) \Rightarrow ([a],[a]) \rightarrow \text{Bit}$

Only works on words.

EXAMPLE 5-27 *Using comparison operators*

The following examples illustrate that equality operators work over arbitrary types, but inequalities are only meaningful for word types. That is, they don't work over bits or higher-dimensioned sequences. Also, the argument types cannot be mixed; they have to have the same type.

```
Cryptol> 2 > 3
False
Cryptol> 2 == 3
False
Cryptol> [2] == [2]
True
Cryptol> [[2 3] [4 5]] != [[3 1] [4 2]]
True
Cryptol> [2] < [3]
["command line", line 1, col 5]:
shape mismatch between [1] and [1][a]

Cryptol> False < True
["command line", line 1, col 7]:
shape mismatch between [a] and Bit

Cryptol> 1 == False
["command line", line 1, col 3]:
shape mismatch between [a] and Bit
```

5.7 Miscellaneous Operators

This section reviews various operators and values in Cryptol that don't quite fit into other categories.

5.7.1 zero

`zero : {a} a`

Gives an arbitrary shaped value whose bits are all False. `~zero` likewise gives an arbitrary shaped value whose bits are all True.

EXAMPLE 5-28 *Using zero*

The value `zero`, unlike `0`, can assume any shape:

```
Cryptol> zero == [False]
True
Cryptol> zero == [[False]]
True
Cryptol> 0 == [False]
True
Cryptol> 0 == [[False]]
["command line", line 1, col 3]:
shape mismatch between [1] and [1][1]
```

A similar comment applies to `~zero`, which is any shape filled with 1 bits.

5.7.2 min

`min : {a} (fin a) => ([a],[a]) -> [a]`

Returns the minimum of its two arguments. The arguments must be words.

EXAMPLE 5-29 *Using min*

```
Cryptol> min (2, 3)
2
Cryptol> min (12, 0)
0
Cryptol> min ([1], [2])
["command line", line 1, col 1]:
shape mismatch between [1] and [1][a]
```

5.7.3 max

`max : {a} (fin a) => ([a],[a]) -> [a]`

Returns the maximum of its two arguments. Complementary to the function `min` above.

5.7.4 parity

parity : {a} (fin a) => [a] -> Bit

Returns the parity of the word (True if there's an odd number of True bits in the word).

EXAMPLE 5-30 *Using parity*

```
Cryptol> parity 0xF
False
Cryptol> parity 0xD
True
```

5.7.5 error

error : {a b} [a][8] -> b

Causes the Cryptol program to fail with the given error message.

EXAMPLE 5-31 *Using error*

Assume the following function is in scope:

```
f x = if x < 10 then x + 1
      else error "f needs an argument less than 10";
```

Then:

```
Cryptol> f 1
0x2
Cryptol> f 12
f needs an argument less than 10
```

See Section 5.7.6, "undefined," below.

5.7.6 undefined

undefined : {a} a

If evaluated, causes the Cryptol program to fail. undefined is very useful for stubbing out incomplete definitions. The value undefined is similar to error; it is useful when no special message is needed.

EXAMPLE 5-32 *Using undefined*

```
Cryptol> if False then undefined else 1
0x1
Cryptol> if True then undefined else 1
Entered Cryptol 'undefined' value
```

See also Section 5.7.5, "*error*," above.

5.8 Format: Pretty printing

The format function in Cryptol provides a simple, yet powerful, way of pretty printing your results. (The operation of format is similar to C's printf.) Let's consider a few examples to see format in operation:

```
Cryptol> :print (format ("<%x, %X, %d, %o, %b>", 12, 12, 12, 12, 12))
<c, C, 12, 14, 1100>
```

```
Cryptol> :print (format ("Sequence: %[] .4x", [1 2 3]))
Sequence: [...1 ...2 ...3]
```

As seen, format is a variable-arity function: unlike other Cryptol functions, format can deal with any number of arguments following the initial format string. (Of course, the format string and the arguments should match, as we will see shortly.)

The first argument to format is the format string, composed of literals and format specifiers. Literals print as they look; for instance in the example above, the literal "Sequence: " prints as itself in the output. (Note the final space in the end; spaces in format strings are significant.)

The acceptable specifiers and their meanings are similar to their C counterparts:

TABLE 5-1 *Format specifiers for pretty printing.*

	Print as...	Argument Type	If the value is	the output is...
B	Boolean	Bit	True	True
b	binary	Cryptol word	123	1111011
o	octal	Cryptol word	123	173
d	decimal	Cryptol word	123	123
x	hexadecimal (a-f)	Cryptol word	123	7b
X	hexadecimal (A-F)	Cryptol word	123	7B

The format of the argument does not affect the form of the output. For example, arguments expressed in octal can be printed in hexadecimal, and vice versa:

```
Cryptol> :print (format ("%x", 0o7777))
fff
```

```
Cryptol> :print (format ("%o", 0xfffff))
177777
```

Format does not put in base prefixes by default, but these may be added as literals in the format string:

```
Cryptol> :print (format ("0x%x", 243))
0xf3
```

5.8.1 Width

Each specifier can be given a width modifier, requiring the result to be fit into a certain number of spaces. For example:

```
cryptol> :print (format ("Free: <%b>, Fixed: <%4b>", 1, 1))
```

```
Free: <1>, Fixed: <   1>
```

Since the first specifier is simply %b, Cryptol uses as many spaces as it needs to print the corresponding argument. In this case, the number 1 requires (in binary) only 1 space to be printed. (Note that defaulting is at work here: at the top level, the number 1 is defaulted to require just a single bit, as that is the least amount of space we need.) However, the second specifier states that the argument should print in exactly 4 spaces, and hence Cryptol leaves 3 blank spaces before printing the number 1.

What happens if the number does not fit in the specified number of spaces? Here is the response from Cryptol:

```
cryptol> :print (format ("<%1x>", 17))  
["command line", line 1, col 1]: size 1 < 2!
```

The type checker complains that 17 is too large a number to fit into one space for hexadecimal printing, and the expression is rejected. Cryptol does *not* extend the required space to accommodate the larger value. The width is absolute.

Special warning about the %d specifier: It may surprise you to find out that certain numbers will be rejected, even though they can fit into the space provided with the %d specifier:

```
cryptol> :print (format ("<%2d>", 63))  
<63>
```

```
cryptol> :print (format ("<%2d>", 64))  
["command line", line 1, col 1]: size 2 < 3!
```

Given the specifier %2d, Cryptol deduces that it can accommodate an argument up to 6 bits. It would be able to handle *some* 7-bit arguments in two spaces, say up to decimal value 99, but *not all* 7-bit arguments, such as decimal value 123. So it is not safe to allow any 7-bit arguments to be accepted.

When checking the type of a format expression, the format string determines the types of all of the expected arguments. The actual arguments are then checked for conformance with those types. So the type of the actual argument cannot influence the type expected by the format string. So despite the fact that we can observe that 64 can be printed in two spaces, Cryptol can only rely on the type specification to make this decision. This is not a problem for any other bases, since the length of a printed representation for binary, octal or hexadecimal will fully correspond to the length of its type.

5.8.2 Padding

Given that we can specify a field width, it is natural that we may want to control the pad character used when the actual value is shorter than the specified width. The default pad character is the blank. To specify the pad character, precede the width with the padding string in curly braces:


```
Cryptol> :print (format ("<%.10x>, <{%0}10x>, <{%0x0}10x>",
                        0xABCD, 0xABCD, 0xABCD))
<.....ABCD>, <000000ABCD>, <0x0000ABCD>
```

If multiple characters appear in the padding string, as is done above with 0x0, the last character will repeat to fill the field. (If the multiple-character padding string is too long for the space available, it will be truncated, starting from the right hand side, to fit.)

As a shortcut notation, if the padding string is a single character, the curly braces may be omitted:

```
Cryptol> :print (format ("<%.10x>, <%010x>,
                        {%0x0}10x>", 0xABCD, 0xABCD, 0xABCD))
<.....ABCD>, <000000ABCD>, 0x0000ABCD>
```

5.8.3 Printing tuples

The following example demonstrates how tuples can be formatted:

```
Cryptol> :print (format ("%(x,d)", (12,12)))
(c,12)
```

The specifier `%(_,_)` indicates that we will be formatting a 2-tuple. Similarly, a 3-tuple would be specified by `%(_,_,_)` and so on for arbitrary n-tuples. The underscores are filled with new format strings (without enclosing quotations). That allows us to put literals, line breaks and other formatting information in between tuple elements. And tuple formatting can handle nested tuples as well:

```
Cryptol> :print (format ("%(a.1: %x, a.2: %x), b: %x)", ((1,2),3)))
((a.1: 1, a.2: 2), b: 3)
```

Not all elements of the tuple need to be printed:

```
Cryptol> :print (format ("%(ignore the first one, %d)", (1,2)))
(ignore the first one, 2)
```

Of course, there can be at most one specifier for each element, otherwise Cryptol will complain (for instance, the format string `%(d:%x, %d)` will be rejected.)

5.8.4 Printing sequences

The format function provides special support for pretty printing sequences. Recall that a Cryptol word is simply a sequence of bits. Hence, we can use the `%B` specifier to get the individual bits of a number:

```
Cryptol> :print (format ("%[]B", 12))
[False False True True]
```

Of course, we can print non-bit sequences as well:

```
Cryptol> :print (format ("%[]x", [1 2 3 4]))
[1 2 3 4]
Cryptol> :print (format ("%[][]x", [[1 2] [3 4]]))
[[1 2] [3 4]]
Cryptol> :print (format ("%[][][]x", [[[1 2] [3 4]]]))
[[[1 2] [3 4]]]
```

The depth of the sequence is specified in the format string. Each pair of []'s indicates another level of nesting.

Padding and width specifiers can be used in the sequence format specifier:

```
Cryptol> :print (format ("%[] [] {0x0}8x", [[1 2] [3 4]]))  
[[0x000001 0x000002] [0x000003 0x000004]]
```

To control the printing of sequences across lines:

```
Cryptol> :print (format ("%[1] [] {0x0}8x", [[1 2] [3 4]]))  
[[0x000001 0x000002]  
 [0x000003 0x000004]]
```

The format string %[1][]{0x0}8x states that we want to print a two-level sequence, each element occupying 8 spaces, padded starting with 0x0, and filled with 0's. Furthermore, the modifier [1] states that we want a line break after each element at that level.

We can use the line break specifiers at multiple levels:

```
Cryptol> :print (format ("%[2] [3] {0}3x",  
 [[1 2 3 4 5 6 7 8] [7 8 9 0 1 2 3 4] [3 4 5 6 7 8 9 0] [9 0 1 2 3 4 5  
 6]]))  
  
[[001 002 003  
 004 005 006  
 007 008]  
 [007 008 009  
 000 001 002  
 003 004]  
  
 [003 004 005  
 006 007 008  
 009 000]  
 [009 000 001  
 002 003 004  
 005 006]]
```

The darker shading indicates the 3 elements separated by a line break as controlled by the inner [3] specifier. (There is a final line break after the last two elements at that level.) The lighter shading indicates the two elements separated by a line break at the outer level.

Format properly indents layers so that elements will line up as they occur on separate lines.

Cryptol allows custom separators for printing sequence elements. The separator string follows a forward slash at the level the separator will be used:

```
Cryptol> :print (format ("%[/;] [/,]d", [[1 2] [3 4]]))  
[[1,2];[3,4]]
```

Inner elements are separated by a ',' and outer elements by a ';'. Element separators can be used with line breaks as well:

```
Cryptol > :print (format ("%[1/ EOL] [/.;.]d", [[1 2] [3 4] [5 6]]))  
[[1.;.2] EOL  
 [3.;.4] EOL  
 [5.;.6]]
```

The following eliminates the space as a default element separator:

```
Cryptol > :print (format ("%1/][/]d", [[1 2][3 4][5 6]]))  
[[12]  
 [34]  
 [56]]
```

Here is a final example demonstrating various format options:

```
cryptol> :print (format ("%4/,](%{0x0}4x,%{0x0}4x)", [| (x,x) || x <-  
 [1..16] |]))  
[(0x01,0x01), (0x02,0x02), (0x03,0x03), (0x04,0x04),  
 (0x05,0x05), (0x06,0x06), (0x07,0x07), (0x08,0x08),  
 (0x09,0x09), (0x0a,0x0a), (0x0b,0x0b), (0x0c,0x0c),  
 (0x0d,0x0d), (0x0e,0x0e), (0x0f,0x0f), (0x10,0x10)]
```


6

Cryptol Idioms

Learning how to use a new language usually involves learning its idioms. In this chapter, we will describe several basic Cryptol idioms that we have identified.

6.1 Padding

It is very common to pad data to a larger size. In Cryptol, we can take advantage of size polymorphism to define padding in a declarative fashion. In the following example, we pad a key on the right with a single 1 bit followed by zeros, and finally a 64-bit sequence containing the size of the key. Notice that the key is constrained to have a size that can be expressed in 64 bits.

```
pad : {a} (64 >= width(a)) => [a] -> [512];
pad key = key # [True] # zero # sz
  where {sz : [64];
        sz = width key;
        };
```

The zero in the middle is unconstrained in size, while everything else is of a fixed or given size. Thus, in order to make the result be 512 bits wide, the zero will be sized to the appropriate number of 0 bits in order to fill the remaining space.

6.2 For-Loops and Recurrence Relations

In an imperative language, for loops are a fundamental building block of cryptographic algorithms. In Cryptol, we express this control more declaratively by using sequence comprehensions and defining recurrence relations.

EXAMPLE 6-1 For loops (I)

Consider the following imperative loop in C:

```
sum = 0;

for (i = 0; i < 10; i++)
    sum = sum + i;

return sum;
```

In Cryptol we would identify the state (`sum`) and then write a recurrence relation corresponding to this loop. We define the sequence of states (`sums`) with an initial value of 0. To get the final value we index into the sequence of states at the desired location: `sums @ 10`:

```
result = sums @ 10;
sums = [0] # [| sum + i || sum <- sums
              || i <- [0 .. 9]
              |];
```

Here is another example from the SHA1 specification of the compression function:

- ◆ Divide M_i into 16 words: $W_0 \dots W_{16}$, where W_0 is the left-most word.
- ◆ For $t = 16$ to 79, set W_t to $(W_{t-3} \text{ XOR } W_{t-8} \text{ XOR } W_{t-14} \text{ XOR } W_{t-16}) \lll 1$.

In Cryptol, we identify the state as being initialized for the first 16 steps with `m`. Each of the succeeding states is calculated by offsets into the state. For example, to reference a state that occurred 14 steps ago in a 16-entry state, we have to drop the first two elements; i.e. we index the state at an offset of 2: `drop (16- 14, ws)`.

EXAMPLE 6-2 For loops (II)

```
compress1 : [16][32] -> [80][32];
compress1 m = ws @@ [0 .. 79]
  where {ws : [inf][32];
        ws = m # [| (w3 ^ w8 ^ w14 ^ w16) <<< 1
                  || w16 <- drop (16- 16, ws)
                  || w14 <- drop (16- 14, ws)
                  || w8 <- drop (16- 8, ws)
                  || w3 <- drop (16- 3, ws)
                  |];};
```

6.3 Cryptographic Modes

Cryptographic modes are easily described using sequence comprehensions. Electronic Code Book is particularly straightforward:

```
ecb(xs, key) = [| encrypt(x, key) || x <- xs |];
```

CBC mode is expressed as a simple recurrence relation:

```
cbc(iv, xs, key) = ys
  where ys = [| iv | # [| encrypt(x ^ y, key)
                    || x <- xs
                    || y <- ys
                    |];
```


7

Cryptol Tips and Pitfalls

This chapter collects several code snippets, tips, and pieces of advice that might be of interest to Cryptol users in general.

7.1 Understanding the type of an enumeration

Assume we want to replace the i th element of a sequence, for some i , without changing other elements. One way of solving this problem in Cryptol is as follows:

EXAMPLE 7-1 *Replacing the i th element of a sequence*

```
repl (xs, i, y) = [| if k == i then y else x
                  || k <- [0 .. ]
                  || x <- xs
                  |];
```

A couple of experiments:

```
Cryptol> repl ([1 .. 10], 3, 7)
[1 2 3 7 5 6 7 7 9 10]
Cryptol> repl ([1 .. 10], 12, 7)
[1 2 3 4 5 6 7 8 9 10]
Cryptol> repl ([ [1 2] [3 4] [5 6] ], 1, [7 8])
[[1 2] [7 8] [5 6]]
```

What is the type of `repl`? We expect something along these lines:

$$\{a\ b\ c\} ([a]b, [c], b) \rightarrow [a]b$$

The argument `xs` has type $[a]b$. That is, it is a sequence of length a , where each element has some size denoted by b . The second argument, `i`, has type $[c]$, i.e. it is some word, denoting the index. The final argument, `y`, is the new value for the position, and it should have type b . Given all that, the result simply has type $[a]b$, same as the input sequence `xs`.

The type inferred by Cryptol is quite similar, except it has a predicate on c :

$$\{a\ b\ c\} (c \geq 1) \Rightarrow ([a]b, [c], b) \rightarrow [a]b$$

That is, Cryptol requires the size of the index to be at least 1 bit. At first, this looks over-constraining, as nothing in the definition of `repl` seems to be indicating such a need.

Looking more closely, we see that the parameter `i` is used in the expression `k == i`, and hence the type of `i` and `k` must be the same. (See Example 5-27, "Using comparison operators," on page 73 for a discussion of comparison operators and their types.) What is the type of `k`? It is drawn from the enumeration `[0 ..]`, which is a shorthand for `[0 1 ..]`. That is, each element is obtained from the previous element by adding one. Since we are using the addition operator, Cryptol deduces that `k` should have at least one bit in it, since the literal 1 needs at least one bit to represent. Hence the constraint.

It is important to realize that this constraint might cause trouble if used carelessly at the top level, where literals are defaulted to have the minimum number of bits possible. For instance, consider the expression:

```
[0 .. ] @@ [1 .. 5]
```

We might expect to get the sequence [1 2 3 4 5], but alas, the result really is:

```
[1 0 1 0 1]
```

The reason is that the enumeration [0 ..] is determined to have type $(a \geq 1) \Rightarrow [\text{inf}][a]$, but then immediately is defaulted to the minimum size required, which is $[\text{inf}][1]$. Hence, it is an alternating sequence of 0's and 1's and nothing else!

Defaulting is discussed in Section 2.9, "Defaulting," on page 15.

7.1.1 Manipulating sequences

How can we code sequence multiplication in Cryptol? We want a function that can multiply sequences of compatible sizes. The type we think of is:

```
mmult : {m n k a} ([m][n][a], [n][k][a]) -> [m][k][a]
```

Here, $[a]$ is the size of the elements in these sequences, which are words. The first sequence has size m by n , the second has size n by k , and the final sequence will have size m by k . We assume that the sequences are stored row by row. That is, the typical identity sequence of size 3 by 3 is represented by:

```
[[1 0 0] [0 1 0] [0 0 1]]
```

How do we do multiplication? By definition, the element at position (i, j) in the final sequence is the inner product of the i th row of the first sequence and the j th column of the second. If we have a way of computing the transpose of a sequence, we can define multiplication as:

```
mmult (xss, yss) = [| [| sum (col * row)
                      || col <- transpose yss
                      ||]
                    || row <- xss
                    |];
```

That is, we consider each row of the first argument, and for each row, we consider all columns of the second sequence, which can be obtained as rows of the transposed sequence. Then, we simply use the multiplication operator $*$ over vectors (which is conveniently defined element-wise!) and then sum each row. Although it is builtin to Cryptol, transpose can be defined as:

```
transpose : {m n a} [m][n]a -> [n][m]a;
transpose xss = [| [| xss @ i @ j
                    || i <- [0 .. m]
                    ||]
                 || j <- [0 .. n]
                 |];
  where { n = width (xss @ 0) - 1;
        m = width xss - 1;
        };
```

That is, we simply go over all indices and construct the final sequence accordingly. Notice how we extract the size information in variables m and n .

The final bit of code we need is `sum`, which simply returns the summation of elements of a sequence. Probably the easiest way to do this in Cryptol is by using a recurrence as follows:

```
sum xs = sums @ width xs
where sums = [0] # [| x + y
               || x <- xs
               || y <- sums
               |];
```

Here are a couple of examples:

```
Cryptol> mmult ([1 2 3] [4 5 6]), [[3] [4] [5]])
[[2] [6]]
```

Notice that all arithmetic is modular. The literals in the example all fit into at most 3 bits, so the arithmetic is done modulo 8. To be on the safe side, we need to specify how big our elements are. One way of achieving this is via specifying explicit types for the arguments.

```
Cryptol> mmult (x, y)
  where { x : [2][3][32];
         x = [|1 2 3]
              [4 5 6]];
         y : [3][1][32];
         y = [[3] [4] [5]];
         }
[[26] [62]]
```

(Unless you store the above expression in a file, you'll have to enter it all in one line at the Cryptol prompt. We have used multiple lines for clarity.)

8

Examples

This chapter provides several example Cryptol programs.

8.1 DES

The following Cryptol code defines the core encryption routine for DES.

EXAMPLE 8-1 *DES Encryption*

```
des : {a b} (a >= 7) => ([2**(a-1)], [b][48]) -> [64];
des (pt, keys) = permute (FP, swap (split last))
  where { pt' = permute (IP, pt);
        iv = [ round (k, split lr)
              || k <- keys
              || lr <- [pt'] # iv
              || ];
        last = iv @ (width keys - 1);
      };

round (k, [l r]) = r # (l ^ f (r, k));

f (r, k) = permute (PP, SBox (k ^ permute (EP, r)));

swap [a b] = b # a;

permute : {a b} (b >= 1) => ([a][b], [2**(b-1)]) -> [a];
permute (p, m) = [ m @ (i-1) || i <- p ];
```

8.2 RC5

The following is a complete Cryptol specification for the RC5 cipher.

EXAMPLE 8-2 RC5 cipher

```
// Parameters:

R = 12;
W = 32; // Word size measured in bits
WW = W / 8; // Word size measured in bytes
B = 2 * W; // Block size
BB = B / 8; // Bytes per block
T : [32];
T = 2 * (R + 1); // Length of expanded key

// These two must be PW, and QW, depending on whether W=16, 32, and 64.
// If W is anything else, then see formulas in the paper to how
// to compute them..

Pw = P32;
Qw = Q32;

rc5ks : {a} (32 >= width (4*a), 63 >= a) => [4*a][8] -> [T][W];
rc5ks K
= S @@@ [(v - T) .. (v - 1)]
where
{ LL = width K / WW;
  v = 3 * max (LL, T);

  Si : [T][W];
  Si = take (T, [Pw (Pw + Qw) ..]);
  Li : [LL][W];
  Li = splitBy (LL, join K);

  S = [ | (s + a + b) <<< 3 || s <- Si # S
        || a <- [0] # S
        || b <- [0] # L ];
  L = [ | (l + a + b) <<< (a + b) || l <- Li # L
        || a <- S
        || b <- [0] # L ];

  };

rc5e : {a b} (b >= lg2(b), a >= 0) => ([2*(a+1)][b], [2][b]) -> [2][b];
rc5e (S, PT) = CT
where {
  S' = tail (groupBy (2, S));
  PT' = [(PT@0 + S@0) (PT@1 + S@1)];
  CT = rounds @ (R-1);
  rounds = [ | [a' b'] where { a' = ((a ^ b) <<< b) + s1;
                             b' = ((b ^ a) <<< a) + s2; }
             || [a b] <- [PT'] # rounds
             || [s1 s2] <- S'
             ];
};

rc5d : {a b} (b >= lg2(b), a >= 0) => ([2*(a+1)][b], [2][b]) -> [2][b];
rc5d (S, CT) = PT'
where {
  S' = reverse (groupBy (2, S));
```

```
PT = rounds @ (R- 1);
PT' = [(PT@0 - S@0) (PT@1 - S@1)];
rounds = [| [a' b'] where { a' = ((a - s1) >>> b') ^ b';
              b' = ((b - s2) >>> a) ^ a ; }
          || [a b] <- [CT] # rounds
          || [s1 s2] <- S'
          |];

};

rc5_encrypt(K, PT) = CT
where {
  xK = rc5ks K;
  CT = join (rc5e (xK, splitBy(2, PT)));
};

rc5_decrypt(K, CT) = PT
where {
  xK = rc5ks K;
  PT = join (rc5d (xK, splitBy(2, CT)));
};

// Pre- computed short cut constants:
P16, Q16 : [16];
P16 = 0xb7e1;
Q16 = 0x9e37;

P32, Q32 : [32];
P32 = 0xb7e15163;
Q32 = 0x9e3779b9;

P64, Q64 : [64];
P64 = 0xb7e151628aed2a6b;
Q64 = 0x9e3779b97f4a7c15;
```

8.3 Advanced Encryption Standard

The following is the complete AES encryption example, which was described in Chapter 3.

EXAMPLE 8-3 *AES encryption*

```

Nb = 128 / 32;
Nk = 128 / 32;
Nr = max(Nb, Nk) + 6;

//-----
// Top- level cipher definition

aes : ([4*Nk][8], [4*Nb][8]) -> [4*Nb][8];
aes (Key, PT) = encrypt (xkey, PT)
  where xkey = keySchedule Key;

encrypt (XK, PT) = unstripe (Rounds (State, XK))
  where {
    State : [4][Nb][8];
    State = stripe PT;
  };

stripe : [4*Nb][8] -> [4][Nb][8];
stripe block = transpose (split block);

unstripe : [4][Nb][8] -> [4*Nb][8];
unstripe state = join (transpose state);

//-----
// The AES Rounds

Rounds (State, (initialKey, rndKeys, finalKey))
= final
  where {
    istate = State ^ initialKey;
    rnds = [istate] # [ Round (state, key)
                      || state <- rnds
                      || key <- rndKeys ];
    final = FinalRound (last rnds, finalKey);
  };

last xs = xs @ (width xs - 1);

Round : ([4][Nb][8], [4][Nb][8]) -> [4][Nb][8];
Round (State, RoundKey) = State3 ^ RoundKey
  where {
    State1 = ByteSub State;
    State2 = ShiftRow State1;
    State3 = MixColumn State2;
  };

FinalRound (State, RoundKey) = State2 ^ RoundKey
  where {
    State1 = ByteSub State;
    State2 = ShiftRow State1;
  };

```

```

ByteSub : [4][Nb][8] -> [4][Nb][8];
ByteSub state =
  [| Sbox x || x <- row |]
  || row <- state |];

ShiftRow : [4][Nb][8] -> [4][Nb][8];
ShiftRow state =
  [| row <<< 1
  || row <- state
  || i <- [ 0 .. 3 ] |];

MixColumn : [4][Nb][8] -> [4][Nb][8];
MixColumn state =
  transpose [| multCol (cx, col)
  || col <- transpose state
  |];

multCol (cx, col) = join (mmult (cx, split col));

cx = polyMat [ 0x02 0x01 0x01 0x03 ];

polyMat coeff =
  transpose (take (width coeff, cols))
  where cols =
    [coeff] #
    [| cs >>> 1 || cs <- cols |];

//-----
// Key Schedule

keyExpansion : [4*Nk][8] -> [(Nr+1)*Nb][4][8];
keyExpansion key = W
  where {
    keyCols : [Nk][4][8];
    keyCols = split key;
    W = keyCols # [| nextWord (i, old, prev)
    || i <- [Nk .. ((Nr+1)*Nb-1)]
    || old <- W
    || prev <- drop (Nk-1, W)
    |]; };

nextWord : ([8],[4][8],[4][8]) -> [4][8];
nextWord (i, old, prev) = old ^ prev'
  where prev' =
    if i % Nk == 0 then
      subByte (prev <<< 1) ^ Rcon (i / Nk)
    else if (Nk > 6) & (i % Nk == 4) then
      subByte prev
    else prev;

subByte p = [| Sbox x || x <- p |];

Rcon i = [(gpower (<| x |>, i - 1)) 0 0 0];

keySchedule : [4*Nk][8] -> ([4][Nb][8], [Nr-1][4][Nb][8], [4][Nb][8]);
keySchedule Key = (rKeys @ 0, rKeys @@ [1 .. (Nr-1)], rKeys @ Nr)
  where {
    W : [(Nr+1)*Nb][4][8];
    W = keyExpansion Key;
    rKeys : [Nr+1][4][Nb][8];
    rKeys = [| transpose ws || ws <- split W |];
  };

// Sbox
Sbox : [8] -> [8];

```

```

Sbox x = sbox @ x;

sbox : [256][8];
sbox = [ affine (inverse x) || x <- [0 .. 255] ];

affine : [8] -> [8];
affine xs = join (mmultBit (affMat, split xs)) ^ 0x63;

affMat = [ 0xf1 ] # [ x <<< 1 || x <- affMat || i <- [1 .. 7] ];

generator = 3;

//-----
// Galois field 2^8

irred = <| x^8 + x^4 + x^3 + x + 1 |>;

gtimes : ([8], [8]) -> [8];
gtimes (x, y) = pmod (pmult (x, y), irred);

gpower (x, i) = ps @ i
  where ps = [1] # [ gtimes (x, p) || p <- ps ];

inverse x = if x == 0 then 0 else find1 (ys, 0)
  where {
    ys = [ gtimes (x, y) || y <- [0 .. 255] ];
  };

find1 : ([256][8], [8]) -> [8];
find1 (xs, i)
  = if xs @ i == 1 then
    |
    else
      find1 (xs, i + 1);

// Matrix multiplication (in Galois Field 2^8)

mmult : {a b c} ([a][b][8], [b][c][8]) -> [a][c][8];
mmult (xss, yss)
  = [ [ dot (row, col) || col <- transpose yss ] || row <- xss ];

dot (as, bs) = sum [ gtimes (a, b) || a <- as || b <- bs ];

sum xs
  = sums @ width xs
  where sums = [0] # [ x ^ y || x <- xs || y <- sums ];

// Special case for matrix of bits

mmultBit : {a b} ([a][8], [8][b]) -> [a][b];
mmultBit (xss, yss)
  = [ [ dotBit (row, col) || col <- transpose yss ] || row <- xss ];

dotBit : ([8], [8]) -> Bit;
dotBit (as, bs) = parity [ a & b || a <- as || b <- bs ];

```


Index

- (Subtraction) 54

Symbols

63, 72

! (Select from the end) 67

!! (Reverse Permute) 67

!= (Not equals) 72

(Concatenate) 67

% (Remainder) 55

& (Conjunction) 59

* (Multiplication) 55

** (Exponentiation) 56

+ (Addition) 54

/ (Division) 55

= (Equals) 72

> (Greater than) 72

>= (Greater than or Equal) 72

>> (Right shift) 63

>>> (Right rotate) 63

@ (Select) 66

@@ (Permute) 66

^ (Exclusive-or) 60

| (Disjunction) 60

~ (Complement) 61

A

Advanced Encryption Standard 43, 95

AES 43, 95

Arithmetic 26, 27, 54, 55, 56, 61

B

Bits 6

Boolean operations 28

C

Comment 23

Comparisons 29

29, 29

!= 29

> 29

>= 29

Conditionals (if-then-else) 30

Cryptographic mode 86

CBC 86

EBC 86

Cryptol modes

bit mode 9, 33

D

Definitions 4

DES 92

drop 70

E

Enumerations 88

Equality (==) 29

error 75

F

False 59

For Loop 85

G

Galois Connections ix

groupBy 69

I

Indexing 9

big-endian 9

little-endian 9

J

join 68

L

lg2 (Logarithm base 2) 56

Logical operators 59, 60, 61

M

max 74

min 74

N

negate (Two's complement) 61

Numbers 7

binary 7

decimal 7

hexadecimal 7

octal 7

setting output base 31

P

parity 75

Polymorphism 11, 13

shape 12, 15

size 11, 15

Polynomials 27, 58

R

RC5 93

Recurrence relation 85

Recursion 35

reverse 71

Rotating 31, 63, 64

31, 63

>>> (Right) 63

>>> (right) 31

S

Sequences 8, 66, 89

append 32

comprehensions 34, 85

concatenate 67

dropping 70

Enumeration 8

Indexing 67

indexing 9, 66

infinite 8

join 33

joining 68

multiple indexing 9

multiplication 89

parallel generators 34

Permutation 67

permutation 66

recurrences 35, 85

recursion 35

split 33

splitting 67, 68, 69

taking 69

transpose 90

transposing 71

width 66

Shifting 31, 63, 64

31, 63

>> (Right) 63

>> (right) 31

Size 11

literals 11

split 67

splitBy 68

Streams 35

T

tail 70

take 69

transpose 71

True 59

Tuples 10

Type 7

-- variable 13

annotations 14

declaration 7, 13

defaulting 15, 88

inf 35

signature 13

width 12

U

undefined 75

W

width 66

Z

zero 74
