GALOISCONNECTIONS

purely functional

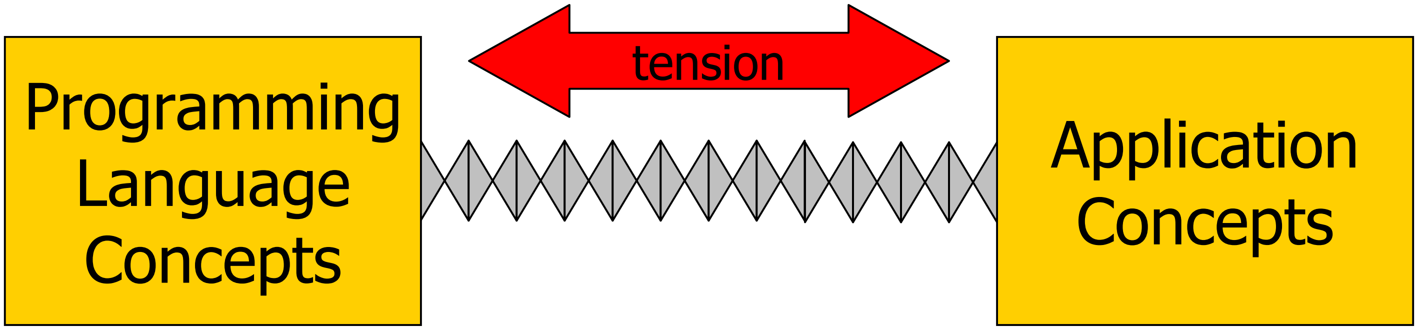
Cryptol Tutorial

Overview and Elements

Purpose of the tutorial

- Overview of Cryptol
  - Domain-specific language for crypto-algorithms
  - Elements of Cryptol
- Cryptol in use for full crypto-algorithm
  - IDEA
  - Idioms
- Non-goal
  - Principles and technology underlying the Cryptol compiler — where does all the code come from?

Tension



- Domain-specific languages (DSLs) attempt to bridge this semantic gap
- Programs are written in domain-specific terms
- Programs “execute” as if a program had been written

Classic examples

- Spreadsheets: Accountancy concepts and notations
- Parser generators: LEX, YACC: use BNF grammars
- SQL: Relational database queries
- PERL: Text manipulation scripting
- TeX and LaTex: Document layout
- Postscript: Low-level graphics
- Mathematica / Maple: Symbolic computation

Value of DSLs

- Design-level programming
  - Huge productivity increase
  - Major flexibility for code evolution
  - Natural maintenance of design documents

- Multiple use
  - Code
  - Test generation
  - Analysis

Where do DSLs come from?

- Domain analysis
  - Existing domain notations
    - Textual
    - Mathematical
    - Graphical
- Semantics must be precise
  - Prototype interpretation must match compiled interpretation must match testing interpretation etc.
  - Source level reasoning
    - DSL programmers may not understand traditional programming

How do domain experts talk to each other?

Crypto-algorithm domain analysis

- What we did
  - Spoke with crypto-algorithm designers
    - What are the important elements of algorithm specification?
  - Studied five AES finalists and DES
    - What do these algorithms have in common?
    - What differences occur between them?
  - Examined previous attempts at crypto DSL
- Now
  - Embodying the domain analysis within a language
  - Obtaining feedback from crypto specialists

Requirements on Cryptol

- Focus on symmetric-key block algorithms
- Emphasize cryptographic concepts and abstractions
  - Rather than traditional computing abstractions
- Specifications expressed in Cryptol should lend themselves to mathematical analysis
  - Capture mathematical structure rather than detailed representation issues
  - Issues of space use and memory allocation should not cloud the specification of crypto algorithms

Consequently, Cryptol should provide:

- Crypto data-types plus operators
  - Fixed-width words, small matrices and vectors, Galois fields
  - Flexible views of data wthout excessive annotations by the user
- Ability to express parameterized algorithms
  - Work over varying key sizes and varying block sizes
- Appropriate control structures for crypto-algorithms
  - Including iteration and recurrence
- Predefined standard cryptographic modes
  - Plus facilities for describing new modes

Block Ciphers

- Logical pattern of use

```
encrypt : (Xkey,PT) -> CT
decrypt : (Xkey,CT) -> PT
keySchedule : Key -> Xkey
```

- In the context of different modes
  - ECB, CBC, etc…

Bit Vectors

- Sizes ranging from 4 bits to 128 bits
  - 8 and 32 most common
- All the usual boolean ops
- Simple modulo arithmetic (+, -, *, /)
- Permutations
  - Mostly just rotations of bit vectors
  - More general permutation used in DES
- Splitting and combining bit vectors
  - Big and little endian

Matrices

- Matrix/vector multiplication used in TwoFish and Rijndael

$$
\begin{aligned}
x_i &= \left\lfloor X/2^{8i} \right\rfloor \bmod 2^8 \qquad i = 0, \ldots, 3 \\
y_i &= s_i[x_i] \qquad i = 0, \ldots, 3
\end{aligned}
$$

$$
\begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} . & \cdots & . \\ \vdots & \mathrm{MDS} & \vdots \\ . & \cdots & . \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}
$$

$$
Z = \sum_{i=0}^{3} z_i \cdot 2^{8i}
$$

- Other arithmetics
  - Polynomials
  - Galois fields

Iteration

- For loops
  - Over fixed counts
- Feistel networks (TwoFish)
- SP-networks (MARS, RC6, ...)
- Recurrence relations

Common practice

- Use of arrays and in-place update
  - Not fundamental to the domain
  - Common implementation technique
  - Underlying model is recurrence relations

$$
\begin{aligned}
(F_{r,0}, F_{r,1}) &= F(R_{r,0}, R_{r,1}, r) \\
R_{r+1,0} &= \mathrm{ROR}(R_{r,2} \oplus F_{r,0}, 1) \\
R_{r+1,1} &= \mathrm{ROL}(R_{r,3}, 1) \oplus F_{r,1} \\
R_{r+1,2} &= R_{r,0} \\
R_{r+1,3} &= R_{r,1}
\end{aligned}
$$

Cryptol

Cryptol

Domain-specific language for cryptoalgorithms

- Now that we know the domain …

  … what should the domain-specific language be like?

Data in Cryptol

- The smallest elements: Bits
- Everything else is a matrix (a parameterized collection)

```
[False True False True False False True]
```

7 single bits

```
0x4A
```

7 (or more) bits

```
[0x3F 0x02 0x41 0xD8]
```

4 elements, each 8 (or more) bits

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

2 elements, each having 4 elements, each 4 (or more) bits

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

10 elements, each of 4 (or more) bits

Hierarchical Views of Data

- Data can be split into n subparts

```
0x99FAC6F975BABB3EDADD847FC237249F

                split

[0x99FAC6F975BABB3E    0xDADD847FC237249F]

        split                  split

[[0x99FAC6F9 75BABB3E] [0xDADD847F C237249F]]
```

Primitive Operations

- Arithmetic operators
  - Result is modulo the word size of the arguments
  - `+ - * / %`
- Boolean operators
  - From bits, to arbitrarily nested matrices of the same shape
  - `& | ^ ~`
- Shift and rotate operators
  - `<< >> <<< >>>`
- Comparison operators
  - Equality, order
  - `== < <= > >= !=`
- Conditional operator
  - Expression-level *if-then-else*
  - Like C's *a?b:c*
  - Booleans are just bits

Matrices

- Matrix operators
  - Concatenation, indexing, size
  - **`# @ @@ width`**

```
[1..5] # [3 6 8]  =  [1 2 3 4 5 3 6 8]
```

- Zero-based indexing from the left

```
[50 .. 99] @ 10  =  60
```

Matrix Operations

- Logical operations lift to matrices of any size
- Arithmetic operations lift to matrices of words
  - Word = matrix of bits
- Bulk indexing

```
[50 .. 99] @@ [10 .. 20]  =  [60 .. 70]
```

- Permutations

```
[1 .. 4] @@ [1 2 3 0]   =  [2 3 4 1]
[1 .. 4] @@ [3 2 .. 0]  =  [4 3 2 1]
```

Cryptol Definitions

- Definitions of values or of functions

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

- Pattern Matching on Matrices

```
sum4 [a b c d] = a + b + c + d;
```

- Nested definitions

```
f x = [y z]
  where {y = x + 1;
         z = not x};
```

Each definition is assigned a type

Simple Cryptol Size Types

- Data sizes

`Bit`	single bit
`[32]`	32-bit word (same as `[32]Bit`)
`[16][48]`	sixteen 48-bit words

- Functions
  - Form is: *argument -> result*

Examples

```
x : [4];
x = 13;

incr : [32] -> [32];
incr x = x + 1;

f : ([16],[16]) -> [16];
f (x, y) = 2 * x + 3 * y + 1

sum4 : [4][32] -> [32];
sum4 [a b c d] = a + b + c + d;
```

**Same code,
Different size
PRODUCES
Different effect
(different
runtime code)**

Widths

- Often it is important to know the size of a structure at the top level
  - Number of bits in a word
  - Number of rows in a matrix

```
width 0x5f = 7
width [[2 3] [4 5] [6 7]] = 3
```

- Width is approximately the $\log_2$ of a number

Equational Correspondence

$$
\begin{aligned}
(F_{r,0}, F_{r,1}) &= F(R_{r,0}, R_{r,1}, r) \\
R_{r+1,0} &= \mathrm{ROR}(R_{r,2} \oplus F_{r,0}, 1) \\
R_{r+1,1} &= \mathrm{ROL}(R_{r,3}, 1) \oplus F_{r,1} \\
R_{r+1,2} &= R_{r,0} \\
R_{r+1,3} &= R_{r,1}
\end{aligned}
$$

```
round [R0 R1 R2 R3] r = [S0 S1 R0 R1]
  where {[F0 F1] = F (R0,R1,r);
         S0 = (R2 ^ F0) >>> 1;
         S1 = (R3 <<< 1) ^ F1;
        };
```

Bounded Iteration

- Borrowed the comprehension notion from set theory
  - **{ a+b | a** $\in$ **A, b** $\in$ **B}**
  - Adapted to matrices (i.e. sequences)

- Applying an operation to each element

```
[| 2*x + 3  ||  x <- [1 2 3 4] |]
  =  [5 7 9 11]
```

Traversals

- Cartesian traversal

```
[| [x y]  ||  x <- [0..2], y <- [3..4] |]

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

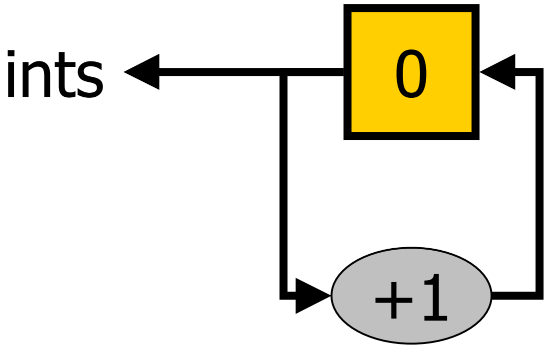
- Parallel traversal

```
[| x + y  ||  x <- [1..3]
          ||  y <- [3..7] |]
  =  [4 6 8]
```

Recurrence

- Textual description of shift circuits
  - Traditionally use a language of commands
    - Arrays, updates, and command-loops
  - Alternatively, use stream-equations
    - Stream-definitions can be *recursive*

```
ints : [inf][256]
ints = [0] # [| y+1 || y <- ints |];
```



Sizes of Recurrences

- Stream is unbounded
  - No finite size
  - But generated by a finite state machine
- Use the inf type

```
ints : [inf][8];
ints = [0] # [| y+1 || y <- ints |];
```

Parallel Traversal

- Define an unbounded sequence of factorials

```
facts : [inf][32];
facts = [1] # [| x * n || x <- facts
                       || n <- [1..] |];
```

Cryptol "Execution"

Execution by calculation

- Example
  - `facts @@ [0..10]`

  is a mathematical expression whose result will be computed
- Storage (and other issues) are left to the Cryptol system

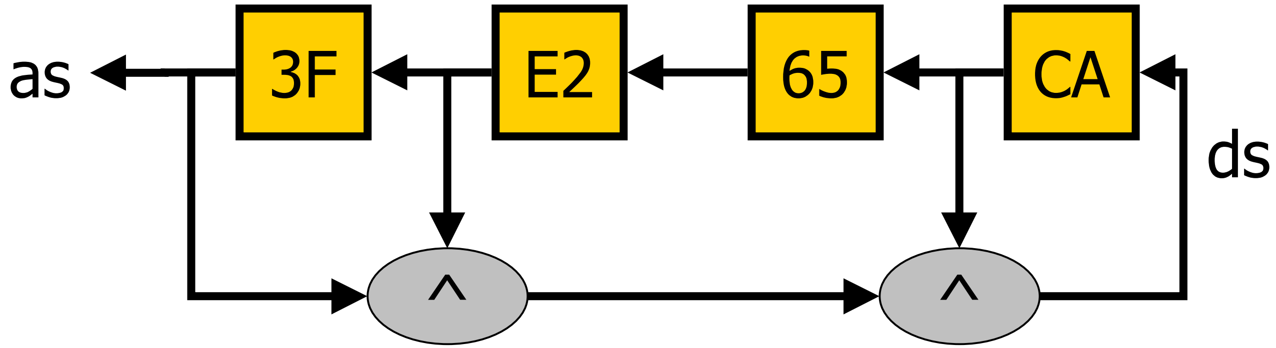
Multiple Access Points

- Define an unbounded sequence of fibonacci numbers, parameterized on starting pair

```
fibs : ([32],[32]) -> [inf][32];
fibs (p,q) = xs
  where
    xs = [p q]
       # [| x + y || x <- xs
                  || y <- drop (1,xs) |];
```

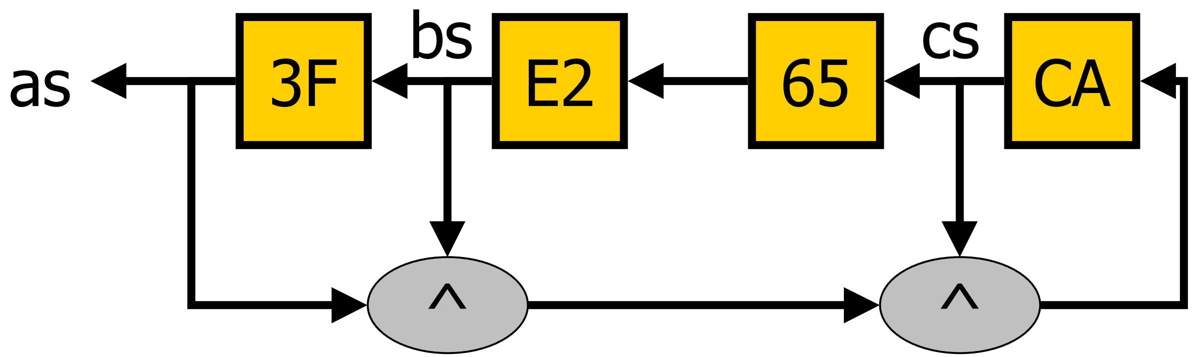
More Complex Stream Equations

```
as = [0x3F 0xE2 0x65 0xCA] # ds;
ds = [| a ^ b ^ c || a <- as
                  || b <- drop(1,as)
                  || c <- drop(3,as) |];
```



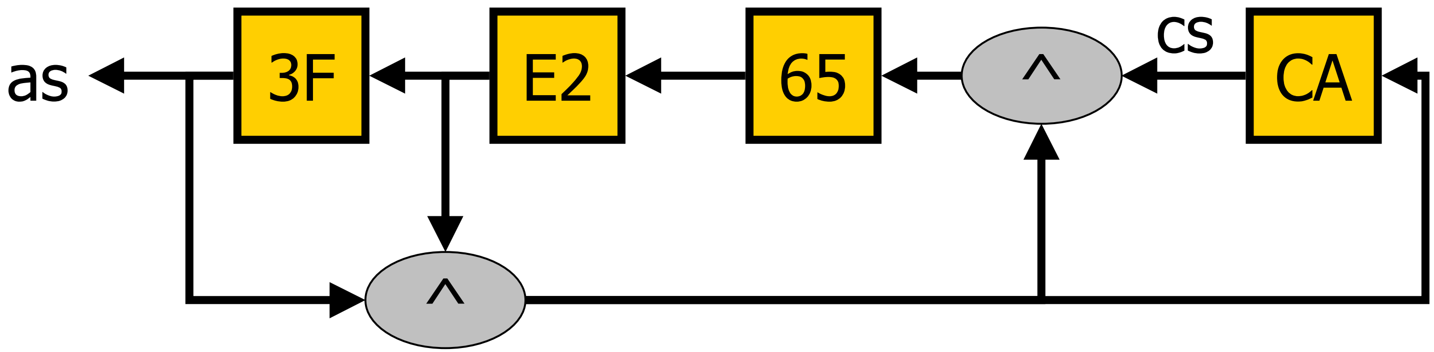
Alternative Description

```
as = [0x3F] # bs;
bs = [0xE2 0x65] # cs;
cs = [0xCA] # [| a ^ b ^ c || a<-as
                             || b<-bs
                             || c<-cs |];
```



Additional Complexity

```
as = [Ox3F OxE2 Ox65]
       # [| c^c' || c <- cs
                 || c'<- drop(1,cs) |];
cs = [OxCA] # [| a^a'
                || a <- as
                || a'<- drop(1,as) |];
```

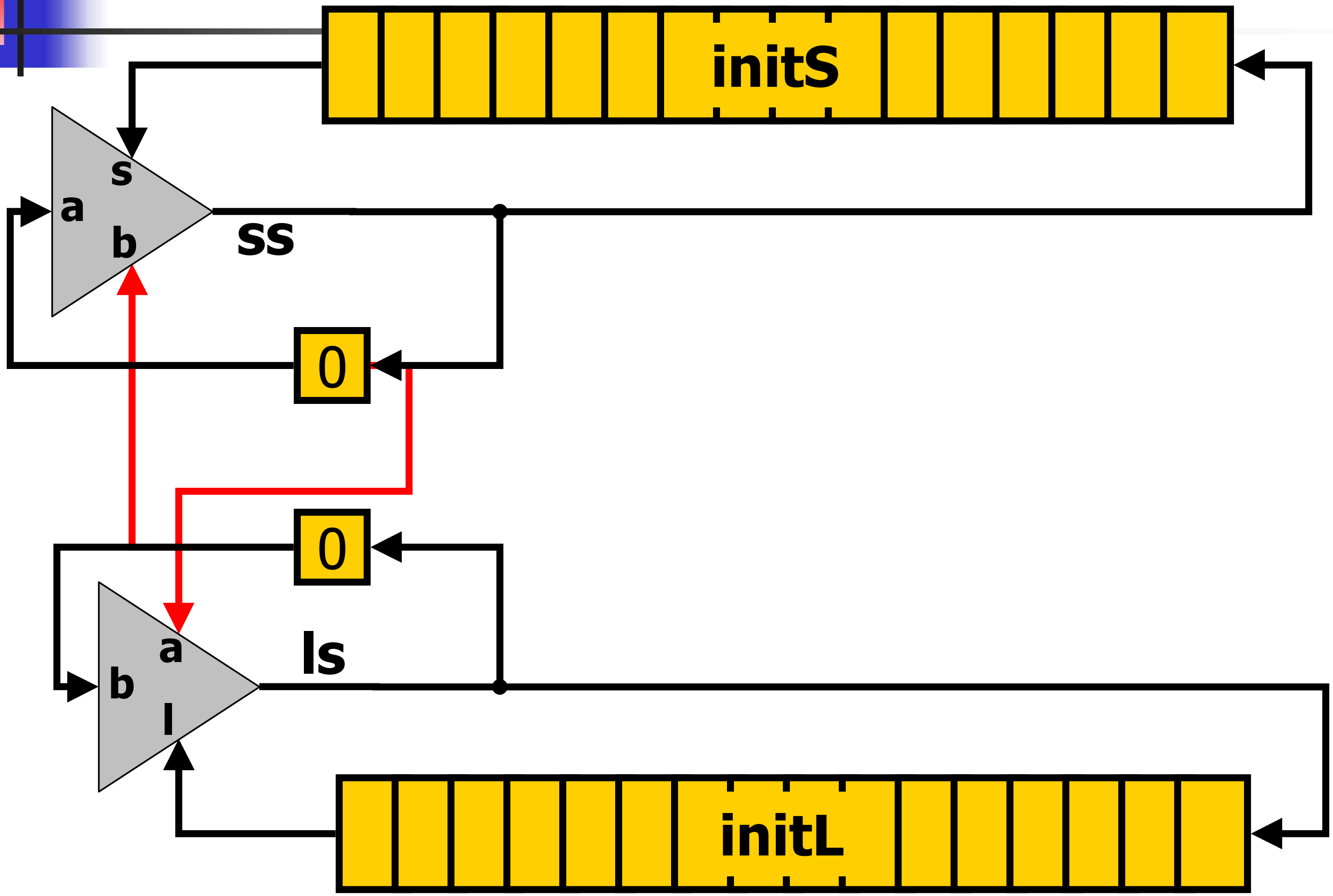


RC6 Key Expansion

- Published specification is in terms of array updates
  - Imperative code for key expansion appears entirely symmetrical — Cryptol exposes non-symmetry

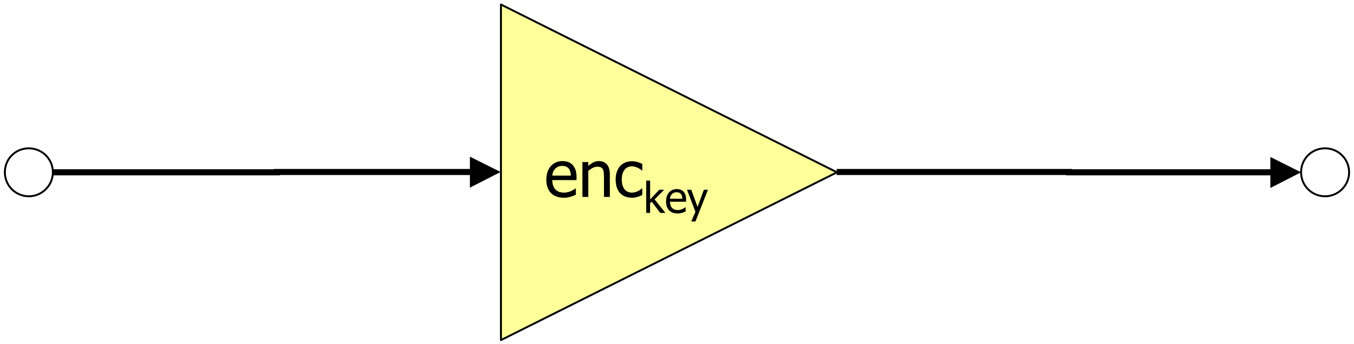
```
ss = [ (s+a+b) <<< 3 || s <- initS # ss
                     || a <- [0] # ss
                     || b <- [0] # ls ];

ls = [ (l+a+b)<<<(a+b) || l <- initL # ls
                       || a <- ss
                       || b <- [0] # ls ];
```

“Circuit” Diagram



Electronic code book

```
ecb(pt, key) = ct
  where
    ct = [| encrypt (x, key) || x <- pt |]
```



Cipher Block Chaining

```
cbc(iv, pt, key) = ct
  where
    ct = [iv] # [| encrypt (x^y, key) || x <- pt
                                      || y <- ct |]
```