Cryptol: A Domain-Specific Language for Cryptographic Service Providers

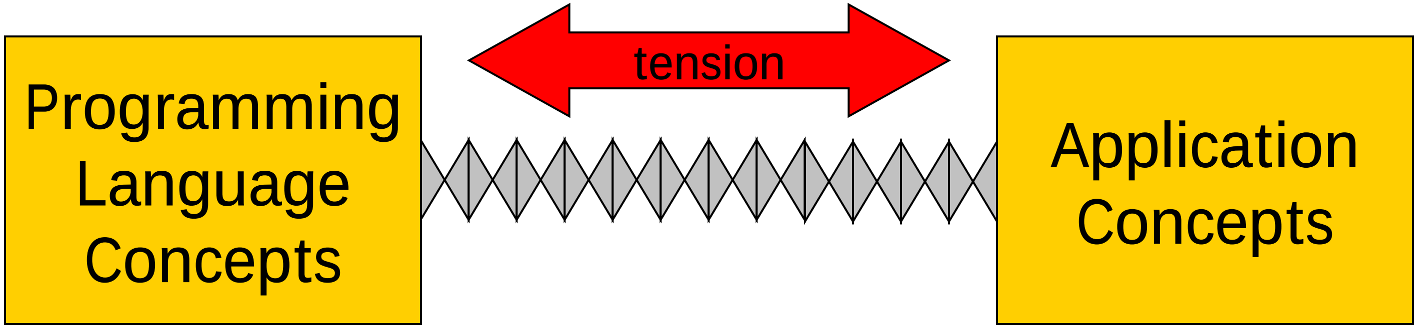
John Launchbury, Jeff Lewis, Thomas Nordin

Galois Connections Inc.

Plan for this Talk

- Why domain-specific languages?
- Domain analysis for crypto-algorithms
- Primitive components of Cryptol
- Intrinsic control structures
- Examples
- Mode specifications

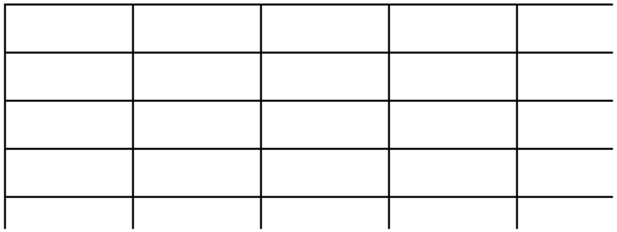
Tension



- Domain-specific languages attempt to bridge this semantic gap
- Programs written in domain-specific terms

Domain-Specific Languages

- Classic examples
  - Spreadsheets
    - Accountancy concepts and notations
  - LEX, YACC
    - Use BNF descriptions of grammars



- Value of DSLs
  - Design-level programming
  - Huge productivity increase
  - Major flexibility in evolvability
  - Natural maintenance of design documents
  - Broadening the programmer base
  - Multiple use: code, test generation, analysis

Where do DSLs come from?

- Existing domain notations
  - Textual
  - Mathematical
  - Graphical
  - Gestural, etc.

How do domain experts talks to each other?

- 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

Crypto-algorithm domain analysis

Cryptol

Domain-specific language for cryptoalgorithms

- Application concepts
  - Data comes in
    - Bits
    - Bit-collections (words)
    - Word-collections, etc.
  - Multiple views of data
  - Equational definitions
  - Bounded iteration
  - Feedback circuits
  - Parameterized definitions

Data in Cryptol

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

7 single bits

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

7 (or more) bits

```
0x4A
```

4 elements, each
8 (or more) bits

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

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

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

10 elements, each
of 4 (or more) bits

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

Hierarchical Views of Data

0x99FAC6F975BABB3EDADD847FC237249F

[0xDADD847FC237249F 0x99FAC6F975BABB3E]

[[0xC237249F 0xDADD847F] [0x75BABB3E 0x99FAC6F9]]

Primitive Operations

- Arithmetic operators
  - Result is modulo the word size of the arguments
- Boolean operators
  - From bits, to arbitrarily nested matrices
- Comparison operators
  - Equality, order
- Conditional operator
  - Expression-level *if-then-else*
- Shift and rotate operators
- Matrix operators
  - Concatenation, indexing, size

Indexing Matrices

- Zero-based indexing from the left

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

  - Numbers are written in traditional notation, but still accessed little-endian

```
0x40 @ 6  =  True
```

- 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

- First-order non-recursive equations

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

- Pattern Matching on Matrices

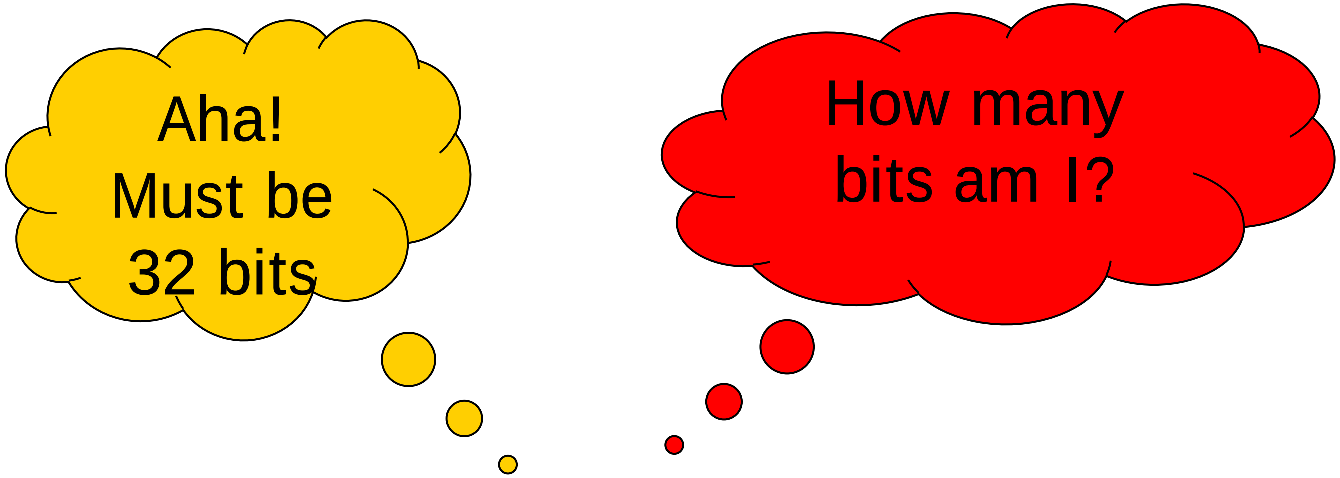
```
sum4 [a b c d] = a + b + c + d
```

- Nested definitions

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

Each definition is assigned a type

Size Polymorphism




```
add32 0xB4 0x3A
```

Size Polymorphism

How many bits am I?

At least 6 bits …

`x = 0x3A`

`x : {a} (a >= 6) => [a]`

Shape Polymorphism

What types do I handle?

Four of something to four of the same thing…

```
swab [a b c d] = [d c b a]
```

```
swab : {a} [4]a -> [4]a
```

Controlling Polymorphism

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

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

Controlling Polymorphism

```
xor : {a} ([a], [a]) -> [a]
xor(xs, ys) = [ (x & ~y) | (~x & y)
                    || x <- xs
                    || y <- ys]
```

A Cryptol Idiom: Padding

Key padding for MD5:

```
pad : {a} (6 >= a) =>
      [a] -> [512*((a+65+511)/512)]

pad key = key # [True] # 0 # size
  where
    size : [64]
    size = sizeOf key
```

0 can have any size, so fills out to satisfy the type constraint

Bounded Iteration

- Borrowed the comprehension notion from set theory
  - **{ a+b | a ∈ A, b ∈ 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]
```

- Cartesian traversal

```
[[x y] || x <- [0..2], y <- [3..4]]
   =  [[0 3] [0 4] [1 3] [1 4] [2 3] [2 4]]
```

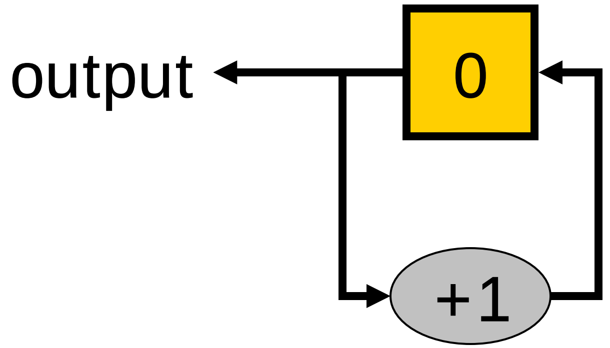
- Parallel traversal

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

Recurrence

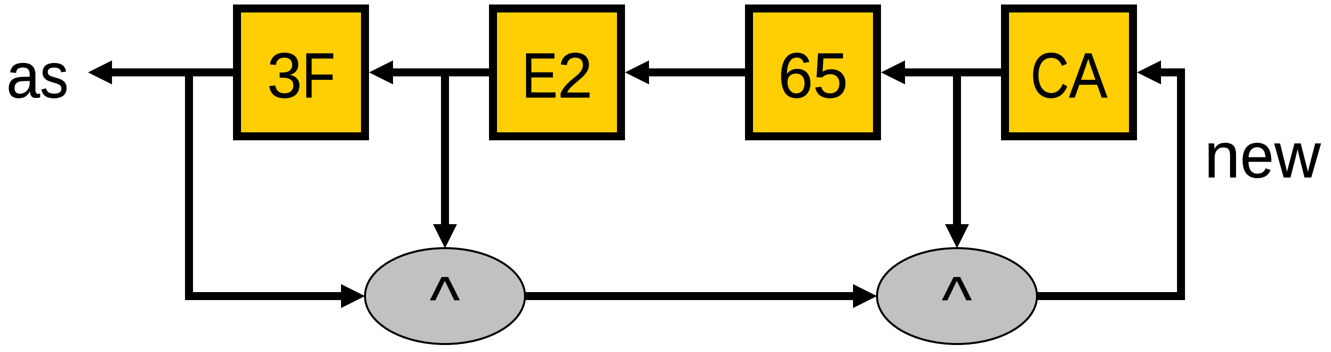
- Textual description of shift circuits
  - Traditionally use a language of commands
    - Arrays, updates, and command-loops
  - Alternatively, use stream-equations
    - Stream-definitions can be *recursive*

```
output = [0] # [y+1 || y<-output];
```



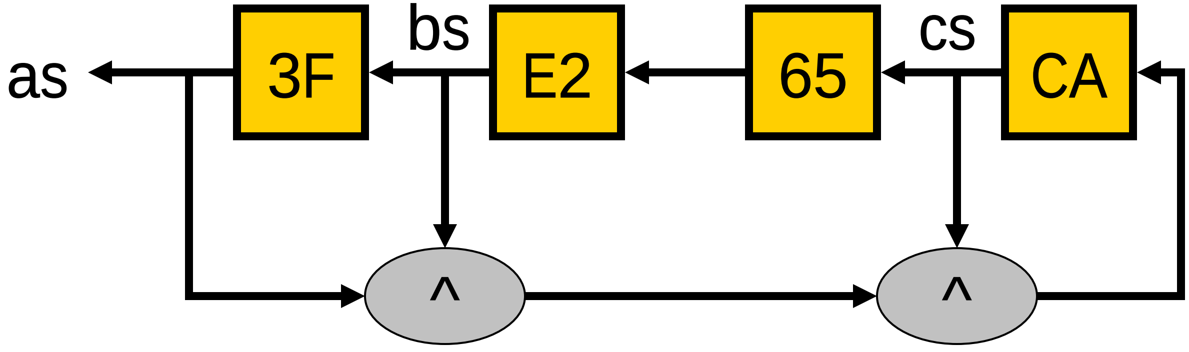
Stream Equations

```
as  = [0x3F 0xE2 0x65 0xCA] # new;
new = [a ^ b ^ c || a <- as
                 || b <- as @@ [1 .. ]
                 || c <- as @@ [3 .. ]];
```



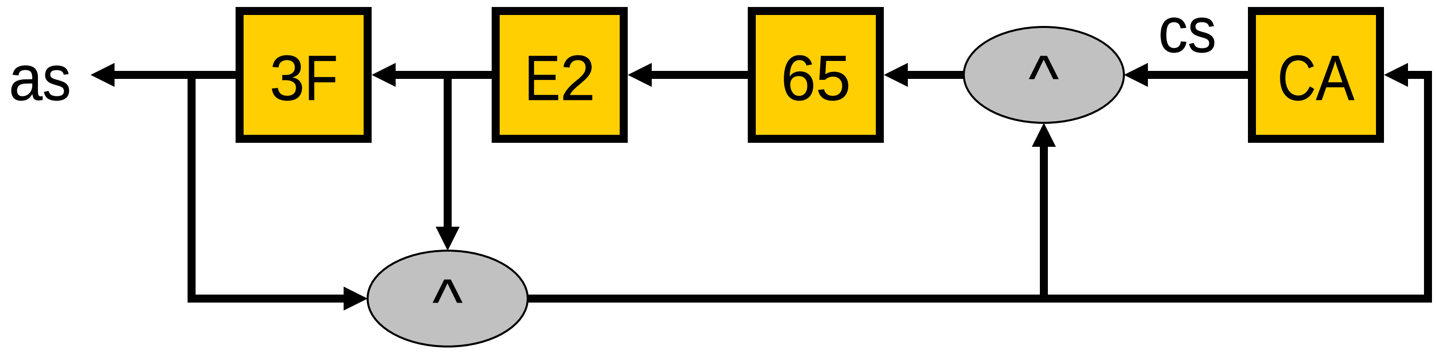
Alternative Description

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



Additional Complexity

```
as = [0x3F 0xE2 0x65]
     # [c^c' || c <- cs
              || c'<- cs @@ [1 .. ]];
cs = [0xCA] # [a^a' || a <- as
                     || a'<- as @@ [1 .. ]];
```



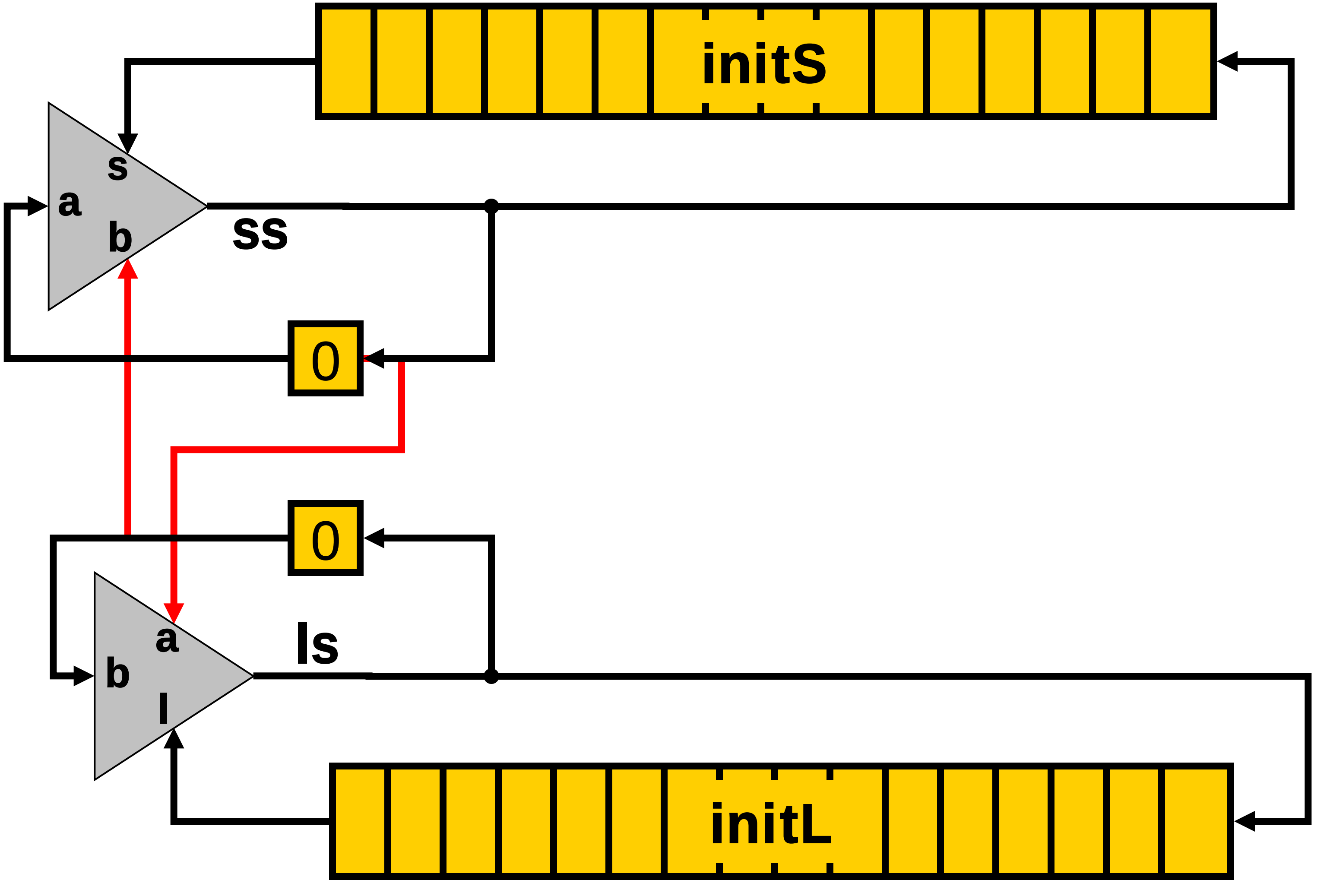
RC6 Key Expansion

- Original specification is written in terms of arrays and updates
  - Key expansion code appears entirely symmetrical
  - Cryptol demonstrates exposes non-symmetry
    - No hidden effects

```
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



Cryptol Idiom: For Loops

- Factors
  - Capture the body of the for-loop as a function
  - Identify the state variables
  - Define a recurrence

- Example
  - Sum the elements of a matrix:

```
sum xs = sums @ (sizeOf xs - 1)
  where sums = [ x + y || x <- xs
                       || y <- [0] # sums ];
```

DES Encryption

```
des (pt, keys) = permute (FP, swap last)
  where
   { pt' = permute (IP, pt);
     iv  = [ round (k, lr) || k  <- keys
                           || lr <- [pt'] # iv ];
     last = iv @ (sizeOf keys - 1);
   };


round (k, [l r]) = r # (l ^ f (r, k))
  where
    f (r, k) = permute(PP, SBox(k ^ permute(EP, r)));
```

DES SBox Lookup

```
SBox : [48] -> [32]
SBox x = join [ sbox (n, b) || n <- [0 .. 7]
                            || b <- split x ];

sbox : ([4], [6]) -> [4]
sbox (n, [b1 b2 b3 b4 b5 b6]) = (s @ n
                                   @ [b1 b6]
                                   @ [b2 b3 b4 b5]);
```

Indexing nested structures.
@ is left-associative

Cryptol Types

- Two kinds of types
  - Value types (Bits, n-Dimensional matrices)

    `Bit      [32]     [a][48]     [6][b]c`

  - Size types (describe the size of matrices)
    - Finite: **`16    a+7    2**(b-1)`**
    - Infinite: **`ko(4)`**
- Definitions have constraints
  - Size constraints: provide lower-bounds on sizes

    `a >= 6       b >= min(7, c + d)`

  - Subtype constraints *(experimental)*:

    `[a*b]c <= [a][b]c`

Current Cryptol Compiler

- Type System
  - Variant of Hindley-Milner style type system
    - Prevents inconsistent use of sizes
    - Identifies large class of ill-formed streams
  - Implementation
    - Constraint-simplification is currently done ad hoc
    - Plan to integrate in an off-the-shelf arithmetic solver
- Execution
  - Interpreter is well developed
  - C-code generator is nearly finished
    - Can then use Cryptol as a crypto-YACC

One Specification, Multiple Implementations

- Fundamental DSL concept:

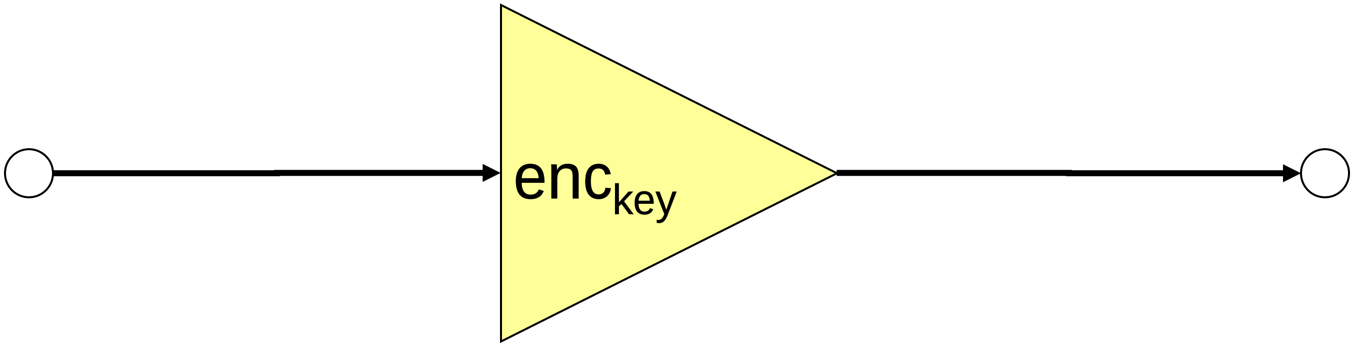
  Distinguish between model and rendition

- Cryptol specifications are designed to be independent of the target language
  - Interpret specification
  - Reference implementation
  - Generate C code or Java
  - Machines with a alternate word sizes
  - Generate AIM code
  - Wrapper to make CDSA compliant

But what about cryptographic modes?

Electronic code book

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



Cipher Block Chaining

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