

Simple Ciphers

Brian Alliet

May 3, 2005

1 Introduction

This is a Haskell implementation of some simple cryptographic ciphers. Encryption, decryption, and cryptanalysis/breaking functions are provided for the following ciphers:

- Shift Cipher - $x \rightarrow (x + \kappa) \pmod{26}$
- Substitution Cipher - map based on a permutation of the alphabet
- Affine Cipher - $x \rightarrow \alpha x + \beta \pmod{26}$
- Vigenère Cipher - shift based on a repeating vector
- The Playfair Cipher - substitute based on a matrix
- The ADFGX Cipher - substitute and transform based on a matrix and a keyword
- The Hill Cipher - multiply the message by a matrix
- The Homework 2 Cipher - a hacked up Vigenère cipher

A simple command line user interface is also provided for all the above functions.

2 Data Types

```
type Language = [Double]
type Cipher a = a → [Int] → [Int]
type Break a = [Int] → Language → [(a, [Int])]
type PBreak a = [Int] → Break a
type ValidKey a = a → Bool
```

The the three major functions of this program are encrypting messages, decrypting cypher-text, and performing cryptanalysis and breaking cypher-texts. Both encryption and decryption take a key and an input (as a string of integers) then mutate the data somehow, and return a new string of integers. This is represented by the Cipher type.

Breaking the encryption on a piece of cypher-text requires two pieces of information, a list of frequencies for each possible value in the cypher-text, and the cypher-text itself. The break function takes these inputs and returns a list of possible plain-text possibilities including the key used to generate it. A known plaintext attack requires the known plaintext in addition to the language and ciphertext.

```
base :: Int
base = 26
```

All the ciphers functions below operate on Ints mod base

```
padding :: Int
padding
  | base == 26 = head $ toNumbers ['x']
  | otherwise = error "padding needs to be specialized for this base"
```

Many block ciphers require the plaintext to be padded to fit evenly into a block. padding is the value to use as the padding. padding **must** be within [0..base-1].

```
toNumbers :: [Char] → [Int]
toNumbers
  | base == 26 = map ((subtract $ ord 'a') . ord . toLower) . filter isAlpha
  | otherwise = error "toNumbers needs to be specialized for this base"

fromNumbers :: [Int] → [Char]
fromNumbers
  | base == 26 = map $ chr . (+ord 'a')
  | otherwise = error "fromNumbers needs to be specialized for this base"
```

It is much easier to deal with messages as lists of numbers internally, however, most messages originate as strings. toNumbers converts a string to a list of integers representing the string. This strips out non-alphabetic characters to fit them into the (mod base) system. fromNumbers converts the list of integers back into text.

3 Stream Ciphers

3.1 The Shift Cipher

```
shiftEncrypt :: Cipher Int
shiftEncrypt = map . (addMod base)
```

The shift cypher encrypts messages by shifting each character in the message by a fixed amount. (a → D, b → E, c → F, etc). This shifting wraps around (so the result is always in (mod base)) so x → A, etc.

```
shiftDecrypt :: Cipher Int
shiftDecrypt = shiftEncrypt . negate
```

Decrypting these messages is simply a matter of shifting the characters back in the other direction ($D \rightarrow a$, etc). It can be thought of as “encrypting” the message using the additive inverse of the key used to encrypt the message originally.

```
shiftBreak :: Break Int
shiftBreak = bruteForceBreak shiftDecrypt [0..base-1]
```

The shift cipher can easily be broken using a brute force attack. More sophisticated attacks are possible but the key space is so small using brute force is perfectly adequate. The key space is simply $[0..base-1]$.

```
shiftValidKey :: ValidKey Int
shiftValidKey = const True
```

Any integer is a valid key for the shift cipher.

3.2 The Substitution Cipher

```
substEncrypt :: Cipher [Int]
substEncrypt = map . (!!)
```

The substitution cipher maps each character of the original plain-text to a new character using a permutation of the alphabet (the key).

```
substDecrypt :: Cipher [Int]
substDecrypt key =
  substEncrypt $ map (fromJust.flip elemIndex key) [0..base-1]
```

Decrypting these messages is simply a matter of “encrypting” them using the inverse of the original key. If the original key was BCA the inverse would be CAB. The inverse is obtained by mapping each possible element to its position in the original key.

The substitution cipher cannot easily be broken by brute force, but with a little knowledge of the characteristics of the message’s original language breaking it is pretty straightforward. The process, however, does need a good deal of human interaction and would be fairly difficult to code. Therefore, a break function is not provided for the substitution cipher.

```
substValidKey :: ValidKey [Int]
substValidKey = and . map (==1) . letterCounts
```

A set of integers containing exactly one of every letter is a valid substitution key.

3.3 Affine Cipher

```
affineEncrypt :: Cipher (Int, Int)
affineEncrypt (a,b) = map $ \x → (a*x+b) `mod` base
```

The affine cipher maps each character of the original plain-text to a new character using an affine function $(\alpha x + \beta)$. To ensure multiple plain-text characters aren't mapped to the same ciphertext character $\text{gcd}(\alpha, 26)$ must equal 1.

```
affineDecrypt :: Cipher (Int, Int)
affineDecrypt (a,b) = map $ \x → (multInvMod base a * (x-b)) `mod` base
```

Decrypting these messages is simply a matter of inverting the affine function. The inversion of $\alpha x + \beta$ is $\frac{1}{\alpha}(x - \beta)$. However, $\frac{1}{\alpha}$ cannot be represented in $(\text{mod } \text{base})$ so we need to find another number to represent the same thing. The multiplicative inverse of α is a number that when multiplied by $\alpha \pmod{\text{base}}$ results in 1. After the multiplicative inverse is found (using `multInvMod`) we can simply use it in place of $\frac{1}{\alpha}$.

```
affineBreak :: Break (Int, Int)
affineBreak = bruteForceBreak affineDecrypt
  [(a,b) | b ← [0..base-1], a ← [x | x ← [0..base-1], gcd base x == 1]]
```

Brute force can be used to break the affine cipher over Z_m for small values of m . When $m = 26$ there are only 312 possible keys. Even with larger values of m brute force is still a viable method of attack, it just takes longer. The possible keys for the brute force attack consist of every possible combination of α and β drawing β from $[0..base-1]$ and α from every element of $[0..base-1]$ where $\text{gcd}(n, 26) = 1$.

```
affineValidKey :: ValidKey (Int, Int)
affineValidKey (a,_) = gcd a base == 1
```

Any (α, β) pair where alpha is relatively prime is a valid affine cipher key.

3.4 Vigenère Cipher

```
vigenereEncrypt :: Cipher [Int]
vigenereEncrypt = zipWith (addMod base) . cycle
```

The Vigenère Cipher uses a vector to shift each plain-text character. The first character of the message is shifted by the value of the first element of the vector, the second shifted by the second element of the vector, etc. The vector is repeated as necessary to accommodate the entire message (the vector can be thought of an infinitely long repetition of the key).

```
vigenereDecrypt :: Cipher [Int]
vigenereDecrypt = vigenereEncrypt . map negate
```

Decrypting these messages is simply a matter of "encrypting" the ciphertext with a vector that contains the additive inverse of each of the original keys. This reverses all the shifts done by the encryption yielding the original message.

Breaking the Vigenère cipher is significantly more complicated than breaking the affine and substitution ciphers. A brute force attack is not possible because of the huge key size. Instead, educated

guesses are made about the key size and the key values using character frequency statistics of the language of the clear-text.

The first step to breaking the Vigenère cipher is finding the most probable key length.

```
vigenerereCoincidences :: (Int → [Int] → [Int]) → [Int] → [Int]
vigenerereCoincidences pp m =
  keySortRev $ map (\x → (shiftOverlap x,x)) [1..10]
  where
    overlap = ((length . filter id).) . zipWith (==)
    shiftOverlap n = overlap m' (drop n m') where m' = pp n m
```

The most probable key length is the displacement of the ciphertext that when compared to the original ciphertext yields the most overlap. This is calculated by shifting the ciphertext to the left by various different amounts, finding the amount of overlap for each shift, and sorting the shifts by the amount of overlap. This works because certain letters show up more frequently than others. Once you shift by the same amount as the length of the key “Es” start matching up with other “Es”, etc.

Once the most probable key length is found the actual values of the key must be found. This is done by a series of calculations based on the frequency distribution of the letters in the ciphertext. The values of each element of the key are each found individually.

```
vigenerereShifts :: Language → [Int] → [Int]
vigenerereShifts lang text =
  keySortRev $ map (\x → (langScore (a x) text,x)) [0..base-1]
  where
    a x = e ++ s where (s,e) = splitAt (length lang - x) lang
```

Calculating the shift at a given position (which is the value of this key element) is simply a matter of finding how much we have to shift the frequency distribution for the language to best fit the frequency distribution of the elements of ciphertext effected by this key element.

The frequency distribution off all the characters in the expected language for the plaintext is A_0 . This distribution shifted to the left by n is A_n . Finding the best shift is simply a matter of calculating the “score” for how well the plaintext matches up to each possible shifted frequency distribution. This score is calculated by langScore (see Section 5).

```
vigenerereKeys :: (Int → [Int] → [Int]) → [Int] → Language → [[Int]]
vigenerereKeys preproc text lang =
  concat $ map keysOfLen $ take 3 $ vigenerereCoincidences preproc text
  where
    keysOfLen len = positionPerms $ map (elements len) [0..len-1]
    elements len pos =
      take 3 $ vigenerereShifts lang $ takeEvery len (drop pos text')
      where text' = preproc len text

vigenerereBreak :: Break [Int]
vigenerereBreak text = map (\x → (x,vigenerereDecrypt x text))
  . vigenerereKeys (const id) text
```

Finally, combining everything above together we get the final `vigenereKeys` function. We find the most likely few key lengths, then find the most likely keys of that length. To find the most likely keys of a given length we find the most likely few shifts at each position then take every possible permutation of those shifts. `preproc` is a hack for the Homework 2 cipher. It modifies the ciphertext based on the key length.

```
vigenereValidKey :: ValidKey [Int]
vigenereValidKey = (≠[])
```

Any non-empty string of integers is a valid Vigenère cipher key.

3.5 Homework 2 Cipher FIXME - Find the real name

The homework 2 cipher is a hacked up Vigenère cipher. The only difference between the two is that instead of simply repeating the key (“abcabcabc...”) the homework2 cipher increments each element of the key by 1 every time it is repeated (“abcbcdcdce...”).

```
hw2Transform :: (Int → Int → Int) → Int → [Int] → [Int]
hw2Transform op n = concat
    . map (\(a,b) → map (op a) b)
    . zip [0..]
    . chunkify n
```

Ciphertexts encoded using the homework2 cipher can be made to look like text encoded with the standard Vigenère cipher by simply preprocessing the text to take into account the additional shifts. First we break up the text into chunks of length n (where n is the key length, or a possible key length in the case of the break function), then we add or subtract 1 to all the elements of the second chunk, 2 to all the elements of the third chunk, etc. This essentially reverses the damage done by the homework2 cipher and allows use the resulting text with an unmodified Vigenère cipher.

The same transformation could also be applied to the key (after it has been run through `cycle`) for the encrypt and decrypt function. This would better match the process described in the homework assignment, however, for consistency with the break function, we opt to apply it to the text.

```
hw2Encrypt :: Cipher [Int]
hw2Encrypt k = vigenereEncrypt k . hw2Transform (addMod base) (length k)
```

`hw2Encrypt` simply hacks up the plaintext and passes it on to the Vigenère encrypt function.

```
hw2Decrypt :: Cipher [Int]
hw2Decrypt k = vigenereDecrypt k
    . hw2Transform (addMod base . negate) (length k)
```

`hw2Decrypt` simple de-hacks up the ciphertext and passes it on to the Vigenère decrypt function.

```
hw2Break :: Break [Int]
hw2Break text = map (\x → (x, hw2Decrypt x text))
                . vigenereKeys (hw2Transform (addMod base . negate)) text
```

hw2Break works exactly like vigenereBreak except the hw2Transform function is used to preprocess the text. vigenereKeys takes a “preprocessor” function that will be applied to the key length and ciphertext before doing any operations on the ciphertext. hw2Transform is used to de-hack up the ciphertext before processing.

```
hw2ValidKey :: ValidKey [Int]
hw2ValidKey = (≠[])
```

Any non-empty string of integers is a valid homework2 cipher key.

4 Block Ciphers

```
keywordMatrix :: [Int] → [[Int]]
keywordMatrix kw
  | base == 26 = toMatrix 5 5
                  $ keywordMatrixFilter
                  $ (nub kw) ++ [x | x ← [0..base-1], not (x `elem` kw)]
  | otherwise = error "need to specialize keywordMatrix for this base"
```

keywordMatrix turns a keyword into a matrix (a 5x5 matrix when base== 26).

```
keywordMatrixFilter :: [Int] → [Int]
keywordMatrixFilter
  | base == 26 = filter (≠9)
  | otherwise = error "need to specialize keywordMatrixFilter for this base"
```

keywordMatrixFilter filters out characters from a message that cannot be encoded with the keywordMatrix above (j in the base== 26 case).

4.1 Playfair Cipher

```
playfairOp :: Bool -> Cipher [Int]
playfairOp rev key = playfairOp' . keywordMatrixFilter
  where
    m = keywordMatrix key
    next = (flip mod $ length m).(if rev then pred else succ)

    playfairOp' [] = []
    playfairOp' [a] = playfairOp' [a,padding]
    playfairOp' (a:b:xs)
      | a==b = if rev
        then playfairOp' xs — invalid, skip it
        else playfairOp' (a:padding:b:xs)
      | ya==yb = m!!ya!!next xa : m!!yb!!next xb : rest
      | xa==xb = m!!next ya!!xa : m!!next yb!!xb : rest
      | otherwise = m!!ya!!xb : m!!yb!!xa : rest
    where
      (xa,ya) = fromJust $ matrixIndex a m
      (xb,yb) = fromJust $ matrixIndex b m
      rest = playfairOp' xs
```

The playfair cipher. [FIXME - Document](#)

```
playfairEncrypt :: Cipher [Int]
playfairEncrypt = playfairOp False . filter (/padding)

playfairDecrypt :: Cipher [Int]
playfairDecrypt = (filter (/padding).) . playfairOp True
```

The playfair encoding and decoding functions simply run `playfairOp` after doing the appropriate padding filtering.

```
playfairValidKey :: ValidKey [Int]
playfairValidKey = const True
```

Any string of integers is a valid playfair cipher key.

4.2 ADFGX Cipher

```
adfgxElements :: [Int]
adfgxElements = toNumbers "ADFGX"
```

`adfgxElements` are the values to use to label the rows and columns of of the ADFGX matrix.

```

adfgxEncrypt :: Cipher ([Int],[Int])
adfgxEncrypt (matrixKey, key') = step2 . step1
  where
    m = keywordMatrix matrixKey
    key = nub key'
    kl = length key
    step1 = map (adfgxElements!!)
            . foldr (\(x,y) r → y:x:r) []
            . (map $ fromJust.flip matrixIndex m)

    step2 i = concat
              $ map snd
              $ (sortBy $ \(a,_) (b,_) → compare a b)
              $ zip key
              $ map (\x → takeEvery kl $ drop x i) [0..kl-1]

```

adfgxEncrypt is the ADFGX encryption function. [FIXME - Document.](#)

```

adfgxDecrypt :: Cipher ([Int],[Int])
adfgxDecrypt (matrixKey, key') = step1 . step2
  where
    m = keywordMatrix matrixKey
    key = nub key'
    kl = length key
    step1 (a':b':xs) =
      case (toIndex a', toIndex b') of
        (Just a, Just b) → m!!a!!b : rest
        _ → rest
      where
        rest = step1 xs
        toIndex = flip elemIndex adfgxElements
    step1 _ = []

    step2 i = concat
              $ transpose
              $ map snd
              $ (sortBy $ \(a,_) (b,_) → compare a b)
              $ getcols (sort key) i
      where
        (mincolsize, extra) = quotRem (length i) kl
        getcols [] [] = []
        getcols _ [] = error "should never happen"
        getcols [] _ = error "should never happen"
        getcols (x:xs) i' =
          (n, take consume i') : getcols xs (drop consume i')
          where
            n = fromJust $ elemIndex x key
            consume = mincolsize + (if n < extra then 1 else 0)

```

adfgxDecrypt is the ADFGX decryption function. [FIXME - Document.](#)

```
adfgxValidKey :: ValidKey ([[Int]],[Int])
adfgxValidKey (_,k) = (k≠[])
```

Any matrix/keyword pair with a non-empty keyword is a valid ADFGX key.

4.3 Hill Cipher

```
hillEncrypt :: Cipher [[Int]]
hillEncrypt m = map (flip mod base)
    . concat.concat
    . map (flip matrixMul m . (:[]))
    . chunkify size
    . pad padding size
  where size = length m
```

The Hill cipher is a block cipher that encrypts messages by multiplying chunks of a message by a square matrix. The message is first broken up into chunks that are the same size as the matrix used to encode the message (padding as necessary). Each chunk is then turned into 1 row matrix and multiplied by the key (a $n \times n$ square matrix). The resulting $1 \times n$ matrix is the encoded value for that chunk.

```
hillDecrypt :: Cipher [[Int]]
hillDecrypt = hillEncrypt . matrixInvMod base
```

To decrypt messages encoded with the Hill Cipher we just have to “encrypt” them with the inverse of the original matrix. This will yield the original message.

Unlike most of the previous ciphers the hill cipher cannot easily be broken by using a ciphertext-only attack. It can, however, be broken pretty easily using a known plaintext attack. `hillPBreak` implements a reliable known plaintext attack.

```
hillPBreak :: PBreak [[Int]]
hillPBreak pt ct = bruteForceBreak hillDecrypt keys ct
  where
    maxn = intSqrt $ min (length pt) (length ct)
    keys = filter (matrixInvertibleMod base)
      $ map calcKey
      $ catMaybes
      $ map possible [1..maxn]
    possible n = find (matrixInvertibleMod base . fst)
      $ map unzip
      $ listPerms n
      $ zip (chunkifyExact n pt) (chunkifyExact n ct)
    calcKey (p,c) = matrixMulMod base (matrixInvMod base p) c
```

When the plaintext is known and of sufficient length we can use it to derive the plaintext. First we break the plaintext and cipher text up into chunks of size n where n is a guess at the size of the key matrix. Next we find every possible combination of n matrices from the chunks. This results in a

set of pairs $n \times n$ matrices where the rows of each of the two matrices correspond to each other. For example. If the plaintext was “abcdefghi” and the ciphertext was “jklmnopqr” then one possible set could be:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \text{ and } \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix}$$

These two matrices have an interesting property. If k denotes the key matrix, p denotes the plaintext matrix, and c denotes the ciphertext matrix then $p \times k = c$. This works because multiplying a matrix by another matrix is the same as multiplying each row individually by a matrix and then stacking the results. We’re just encrypting n chunks at a time rather than one at a time.

Now, given $p \times k = c$ we just need to solve for k to find the key. Multiplying both sides by p^{-1} gets us $p^{-1} \times p \times k = p^{-1} \times c$. This can be simplified to $k = p^{-1} \times c$. Now we have a formula for k . Next we simply have to find a plaintext matrix that is invertable (one where the determinant isn’t 0 and the gcd of the determinant and base is 1) and multiply by the ciphertext matrix. This will get us a possible key. We do this for every possible key size to get a list of keys.

Finally all the possible keys (0 or 1 for each possible key size) are run through `bruteForceBreak` to sort the keys in order of the meaningfulness of the plaintext they produce.

```
hillValidKey :: ValidKey [[Int]]
hillValidKey m = matrixInvertableMod base m
                && matrixSquare m
                && length m ≠ 0
```

Any matrix that is invertable (mod base) is a valid hill cipher key.

5 Language Statistical Functions

Given a vector, A_0 , containing a the frequency distribution on all the possible values in the ciphertext a “score” can be given to a given string of plaintext that indicates the likelihood of it being meaningful plaintext.

```
letterCounts :: [Int] → [Int]
letterCounts t = elems $ accumArray (+) 0 (0,base-1) [(x,1) | x←t]
```

The first step in performing this calculation is finding a vector, v , containing number of times each possible character occurs in the given text.

```
letterFrequency :: [Int] → [Double]
letterFrequency text =
    map (\x → (fromIntegral x)/(fromIntegral $ length text)) v
    where v = letterCounts text
```

Next, a vector, w is calculated by dividing each element of v by the total number of elements in the text. This results in a frequency distribution vector similar to the frequency distribution for the whole language (A_0). If the given text is in fact meaningful in the given language these two vectors should be very similar.

```
langScore :: Language → [Int] → Double
langScore a0 = dotProd a0 . letterFrequency
```

Finally, with these two pieces of information in mind (A_0 and w) we can give the text a “score” based on how well it fits the predicted frequency distribution for letters in the given language. This score is calculated by finding the dot product of w and A_0 . As long as letters that appear frequently in the whole language match up with letters that appear frequently in the text the dot product will be high.

6 Brute Force Breaking Functions

When breaking ciphertexts using brute force every possible key is used to decrypt the ciphertext in the hopes that only one key will yield a meaningful message. Although manually locating this message in the sea of nonsense should be fairly simple, it is tedious.

```
probableKeys :: Cipher a → Language → [Int] → [a] → [a]
probableKeys c l ct = keySortRev . map (\k → (langScore l (c k ct),k))
```

To make this process a little easier each possible message is given a “score” based on the language it is expected to be in. This score is calculated by `langScore` (see Section 5). `probableBreaks` sorts a list of possible decryptions of a message by their score.

```
bruteForceBreak :: Cipher a → [a] → Break a
bruteForceBreak c ks ct l = map (\k → (k,c k ct)) $ probableKeys c l ct ks
```

The brute force method of breaking the cypher-text follows the same pattern for every cipher. `bruteForceBreak` takes a decryption function, a list of possible keys, a language, and a piece of cypher-text and returns all possible plain texts for that cypher-text along with the key used to generate it sorted by the likelihood that is a meaningful message.

7 Frequency Distributions for Various Languages

```
anyLang :: Language
anyLang = map average $ transpose $ [snd x | x ← languages, fst x ≠ "any"]

languages :: [(String, Language)]
languages = [
  — Obtained from _Introduction to Cryptography with Coding Theory_
  — by Trappe and Washington
  ("english", [
    0.082, 0.015, 0.028, 0.043, 0.127, 0.022, 0.020, 0.061, 0.070, 0.002,
    0.008, 0.040, 0.024, 0.067, 0.075, 0.019, 0.001, 0.060, 0.063, 0.091,
    0.028, 0.010, 0.023, 0.001, 0.020, 0.001]),
  — Obtained from http://linguistlist.org/issues/5/5-641.html
  ("dutch", [
    0.081, 0.016, 0.000, 0.058, 0.202, 0.000, 0.035, 0.024, 0.072, 0.000,
    0.025, 0.041, 0.024, 0.107, 0.064, 0.016, 0.000, 0.070, 0.045, 0.072,
    0.021, 0.027, 0.000, 0.000, 0.000, 0.000]),
  ("any", anyLang)]
```

languages contains the frequency distribution for letters in various languages.

8 Cipher Function Lookup Table

```
type CipherFunctions a = (
  String → a ([Int] → [Int]),
  String → a ([Int] → [Int]),
  a (Break String),
  a (PBreak String),
  String)

ciphers :: Monad a ⇒ [(String, CipherFunctions a)]
ciphers = [
```

```
  ("shift", (
    cf shiftEncrypt shiftValidKey readIntegral,
    cf shiftDecrypt shiftValidKey readIntegral,
    return $ bf shiftBreak show,
    fail "This function is not available for this cipher",
    "An integer between 0 and " ++ (show base))),
```

```
  ("subst", (
    cf substEncrypt substValidKey (return.toNumbers),
    cf substDecrypt substValidKey (return.toNumbers),
    fail "This function is not available for this cipher",
    fail "This function is not available for this cipher",
    "A list of " ++ (show base) ++ " unique characters")),
```

```

("affine",(
  cf affineEncrypt affineValidKey
    $ tupleMapM readIntegral . splitOn(==','),
  cf affineDecrypt affineValidKey
    $ tupleMapM readIntegral . splitOn(==','),
  return $ bf affineBreak $  $\lambda(a,b) \rightarrow \text{concat } \$ [show a, ", ", show b]$ ,
  fail "This function is not available for this cipher",
  "A pair of integers in the format alpha,beta")),

```

```

("vigenere",(
  cf vigenereEncrypt vigenereValidKey (return.toNumbers),
  cf vigenereDecrypt vigenereValidKey (return.toNumbers),
  return $ bf vigenereBreak fromNumbers,
  fail "This function is not available for this cipher",
  "A word or string")),

```

```

("playfair",(
  cf playfairEncrypt playfairValidKey (return.toNumbers),
  cf playfairDecrypt playfairValidKey (return.toNumbers),
  fail "This function is not available for this cipher",
  fail "This function is not available for this cipher",
  "A word or string")),

```

```

("adfgx",(
  cf adfgxEncrypt adfgxValidKey $
    return . tupleMap toNumbers . splitOn (==','),
  cf adfgxDecrypt adfgxValidKey $
    return . tupleMap toNumbers . splitOn (==','),
  fail "This function is not available for this cipher",
  fail "This function is not available for this cipher",
  "A pair of words in the format matrix_letters,keyword")),

```

```

("hill",(
  cf hillEncrypt hillValidKey readIntegralMatrix,
  cf hillDecrypt hillValidKey readIntegralMatrix,
  fail "This function is not available for this cipher",
  return $ pf hillPBreak showMatrix,
  "A square matrix in the format [[1,2,3],[4,5,6],[7,8,9]]")),

```

```

("hw2",(
  cf hw2Encrypt hw2ValidKey (return.toNumbers),
  cf hw2Decrypt hw2ValidKey (return.toNumbers),
  return $ bf hw2Break fromNumbers,
  fail "This function is not available for this cipher",
  "A word or string"))

```

```

]
where
  cf c v rf s = do
    x ← rf s
    if v x
      then return $ c x
      else fail $ s ++ " is not a valid key"
  — FEATURE: These are sort of ugly
  bf b t c l = map (\(k,p') → (t k,p')) (b c l)
  pf b t p c l = map (\(k,p') → (t k,p')) (b p c l)

```

ciphers maps cipher names to their function. These functions wrap the real functions with functions that map the key values to and from strings and do error checking.

9 User Interface

```

commands :: [(String,CipherFunctions IO → [String] → IO())]
commands = [
  ("encrypt",λ(f,_,_,_,_) → runCrypt f),
  ("decrypt",λ(_,f,_,_,_) → runCrypt f),
  ("break", λ(_,_,f,_,_) → runBreak f),
  ("pbreak", λ(_,_,_,f,_) → runPBreak f),
  ("help", λ(_,_,_,_,h) → const $ putStrLn $
    "The key is in the format : " ++ h)
]

```

commands maps command names to their actions. Encrypt and decrypt run runCipher on the given cipher function. Break runs runBreak if a break function is available.

```

main :: IO()
main = do
  argv ← getArgs
  case argv of
    (cipher:command:args) → do
      fs ← partialLookup cipher ciphers "cipher"
      action ← partialLookup command commands "command"
      action fs args `catch` λe →
        if isUserError e
          then do
            hPutStrLn stderr $ ioeGetErrorString e
            exitFailure
          else ioError e
    _ → usage
  where
    usage = do
      me ← getProgName
      hPutStrLn stderr $ unlines $ [
        "Usage: " ++ me ++ " cipher cmd arg [file,...]",
        " cipher is " ++ (englishJoin "or" $ map fst ciphers'),
        " cmd is " ++ (englishJoin "or" $ map fst commands),
        " arg is the key or language depending on the command",
        " key formats:" ++ map
          (λ(c,(_,_,_,_),h)) → "    " ++ c ++ ": " ++ h)
          ciphers'
      exitFailure
    where
      ciphers' :: [(String,CipherFunctions IO)]
      ciphers' = ciphers

```

The main function. This processes the command line arguments and does whatever action is specified for the command,

```

runCrypt :: (String → IO([Int] → [Int])) → [String] → IO()
runCrypt _ [] = fail "The crypt commands requires a key"
runCrypt cf (key:files) = do
  s ← cat files
  f ← cf key
  putStrLn $ fromNumbers $ f $ toNumbers s

```

runCrypt runs the specified cipher on the input (files or stdin) and prints the result to stdout. The cipher can be either an encryption or decryption function. They both work the same.

```

runBreakOutput :: Int → [(String,[Int])] → IO()
runBreakOutput _ [] =
    putStrLn "Sorry, no more possible decryptions available"
runBreakOutput n ((k,m):xs) = do
    putStrLn $ "Possible decryption #" ++ (show n) ++ ", key: " ++ k
    putStrLn $ fromNumbers m
    c ← keyPrompt stdin stdout "Does this look right? [ynq]" "ynq"
    case c of
        'y' → putStrLn "Done!"
        'q' → do
            hPutStrLn stderr "Aborting!"
            exitFailure
        'n' → runBreakOutput (n+1) xs
        _   → undefined

```

```

runBreak :: IO(Break String) → [String] → IO()
runBreak _ [] = fail "The break command requires a language"
runBreak _ (_:[]) = fail "The break command cannot be run on stdin"
runBreak mf (language:files) = do
    f ← mf
    lv ← partialLookup language languages "language"
    s ← cat files
    runBreakOutput 1 $ f (toNumbers s) lv

```

runBreak runs the break function. runBreakOutput provides an easy to use interface to the results. Each potential plain-text result is presented to the user one at a time and the user can chose to continue or not. This avoids computing all possible results in the common case that the most probable result is the correct result.

```

runPBreak :: IO(PBreak String) → [String] → IO()
runPBreak mf [language,ptf,ctf] = do
    f ← mf
    lv ← partialLookup language languages "language"
    pt ← readFile ptf
    ct ← readFile ctf
    runBreakOutput 1 $ f (toNumbers pt) (toNumbers ct) lv
runPBreak _ _ = fail $
    "The pbreak command requires a language, " ++
    "a plaintext file, and a ciphertext file"

```

runPBreak works just like runBreak except for doing a known plaintext attack instead of a cipertext only attack.

10 Availability

The latest version of SimpleCiphers.lhs (which contains this document and the Haskell souce) is available under an open souce license from:

<http://darcs.brianweb.net/ritcrypto>