```
(map #(case (mod (+ (* %1 6) 11) 91) 0 1 12 6 (char %)) [10 22 21 10]) (map #(char (mod (+ (* %1 6) 11) 91)) [10 22 21 10]) ("G" "4" "." "G") (map #(let [c (mod (+ (* % 6) 35) 91)] (if (= c 35) \# (char c))) [10 22 21 10]) (map #(char (mod (+ (* % 10 10) 10) 10)) [10 10 10 10]) (->> [3 14 0 14] (map #(char (mod (+ (* % 7) 97) 256)))) (->> [23 1 12 1] (map #(char (mod (+ (* % 19 29) 124) 127)))) (->> [23 1 12 1] (map #(char (mod (+ (* % 19 29) 97) 127)))) (->> [1 12 0 1] (map #(char (mod (+ (* (reduce * [1 % 2 3]) % 31) 96) 127)))) (->> [2 1 13 1] (reduce * [1 2 3]) (* 31) (+ 96) (mod 127) (char)) => (->> [1 1 13 1] (map #(char (+ (* % 7) 96)))) ("g" "g" "»" "g") => (->> [1 1 13 1] (map #(char (+ (* % (reduce * (take % (iterate inc 2)))) 96)))) ("b" "b" "" "" "b") => (->> [1 1 13 1] (map #(char (+ (* (bit-and % 23) (bit-or % 23)) 96)))) ("w" "w" "û" "w") => (defn f [n] (if (<= n 0) 1 (* n (f (- n 1))))) (->> [1 1 13 1] (map #(char (+ (f %) 96)))) ("a" "a" "" "" "a") => (let [m1 [[1 2] [3 4]] m2 [[5 6] [7 8]]] (->> [1 1 13 1] (map #(char (+ (* % (get-in m1 [0 0])) (get-in m2 [0 1]) 96)))) ("g" "g" "s" "g")
```

```
[2, 4, 6, 8] |> Enum.map(&("#{&1 * 2 - 2}#{&1 * 3 - 9}#{&1
* 5 - 25}#{&1 * 7 - 49}" |> byte_size() |> to_string()
<> to_string(&1 * &1 * 7 + &1 + 95)) |> String.to_charlist(
)) |> IO.inspect() elixir -e "[2, 4, 6, 8] |> Enum.map(
&(\"#{&1 * 2 - 2}#{&1 * 3 - 9}#{&1 * 5 - 25}#{&1 * 7 -
49}\" |> byte_size() |> to_string() |> (fn x -> Integer.
parse(to_string(x)) |> elem(0) end).() |> (fn y -> to_-
string(y * y * 7 + y + 95) end).())) |> IO.inspect()" elixir
-e 'input = [97, 109, 97]  coeffs = Enum.reverse(input)
  with acc <- 1..4          |> Enum.map(fn x -> x * x
* x * (coeffs |> Enum.at(0)) + x * x * (coeffs |> Enum.
at(1)) + x * (coeffs |> Enum.at(2)) + (coeffs |> Enum.at(
3)) end)          |> Enum.sum(),       codepoint <- rem(
acc, 1111) + 770,       do: IO.puts(:"\##{codepoint}")
' elixir -e "IO.puts \"#{Enum.map([97, 109, 97, 35], fn
x -> (x + 6) * (x + 4) * (x + 2) * x |> rem(26) |> Kernel.
+(97) end) |> Enum.map(&(:erlang.integer_to_list(&1) |>
List.first())) |> :unicode.characters_to_binary()}\"" elixir
-e "IO.puts \"#{Enum.map([97, 109, 97, 35], fn x -> ((x
+ 6) * (x + 4) * (x + 2) * x) |> rem(26) |> Kernel.+(97)
 |> :erlang.int_to_char end) |> List.to_string()}\"" elixir
-e "IO.puts \"#{Enum.map([1, 2, 3], fn x -> (x + 4) * (
x + 1) * (x - 1) |> rem(26) |> Kernel.-(98)||100 end )
|> :erlang.integer_to_list() #         || [109] ++ Enum.
map([2,3],fn x-> 97+((2*x-7)*rem(13))||99end ) #
     ||[97]|> List.foldl(&List::flatten/1,[]) #
    ||:unicode.characters_to_binary()}\"" # ama elixir
-e "IO.puts \"#{Enum.map([1, 2, 3], fn x -> (x + 4) * (
x + 1) * (x - 1) |> rem(26) |> Kernel.-(98)||100 end )
|> Enum.map(&(:erlang.integer_to_list(&1) |> List.first(
))) |> :unicode.characters_to_binary()}\"" elixir -e "IO.
puts \"#{Enum.map([1, 2, 3], fn x -> (x + 4) * (x + 1)
* (x - 1) |> rem(26) |> Kernel.-(98)||100 end ) |> Enum.
map(&(:erlang.integer_to_list(&1) |> List.first())) |>
:unicode.characters_to_binary()}\"" # Encode function plaintext
= "HELLO" encoded = plaintext |> String.downcase() |> String.
graphemes() |> Enum.map(fn x ->  (x |> String.to_charlist(
) |> hd() |> :erlang.binary_to_integer() |> Kernel.-(97)
) * 8 |> Kernel.+(11)   |> rem(26)   |> Kernel.+(97)
|> Integer.to_charlist() end) |> List.to_string() IO.puts(
```

```
encoded) # Decode function ciphertext = "fthvq" decoded
= ciphertext |> String.graphemes() |> Enum.map(fn x ->
 (x |> String.to_charlist() |> hd() |> :erlang.binary_-
to_integer() |> Kernel.-(97) |> Kernel.-(11))   |> rem(
26 * 8)   |> rem(26)   |> Kernel.+(97)   |> Integer.to_-
charlist() end) |> List.to_string() IO.puts(decoded) #
Encode function plaintext = "HELLO" encoded = plaintext
|> String.downcase() |> String.graphemes() |> Enum.map(
fn x ->   (x |> String.to_charlist() |> hd() |> :erlang.
binary_to_integer() |> Kernel.-(97)) |> Kernel.*(8) |>
Kernel.+(11)   |> rem(26)   |> Kernel.+(97)   |> Integer.
to_charlist() end) |> List.to_string() IO.puts(encoded)
# Decode function ciphertext = "fthvq" decoded = ciphertext
|> String.graphemes() |> Enum.map(fn x ->   (x |> String.
to_charlist() |> hd()    |> Kernel.-(97)    |> Kernel.-
(11)    |> rem(26 * 8)    |> rem(26)    |> Kernel.+(97)
   |> Integer.to_string()    |> :erlang.binary_to_integer(
))   |> Integer.to_charlist() end) |> List.to_string() IO.
puts(decoded) elixir -e '# Encode function plaintext =
"HELLO" encoded = plaintext |> String.downcase() |> String.
graphemes() |> Enum.map(&(&1 |> String.to_charlist() |>
hd() |> (&1 - 97 - 11) |> rem(26) |> (&((&1 + 97))) |>
Integer.to_string() |> <<(&1::utf8)>>)) |> List.to_string(
) IO.puts(encoded) # Decode function ciphertext = "fthvq"
decoded = ciphertext |> String.graphemes() |> Enum.map(
&(&1 |> <<(&1::utf8)>> |> String.to_integer() |> (&1 -
97 + 11) |> rem(26) |> (&((&1 + 97))) |> Integer.to_string(
) |> <<(&1::utf8)>>)) |> List.to_string() IO.puts(decoded)
' ** (CompileError) nofile:6: nested captures are not allowed.
 You cannot define a function using  the capture operator
& inside another function defined via &. Got invalid nested
capture: &(&1 + 97)     (stdlib 4.2) lists.erl:1462: :lists.
mapfoldl_1/3    (stdlib 4.2) lists.erl:1463: :lists.mapfoldl_-
1/3    (elixir 1.14.3) src/elixir_fn.erl:140: :elixir_-
fn.escape/3    (stdlib 4.2) lists.erl:1462: :lists.mapfoldl_-
1/3    (elixir 1.14.3) src/elixir_fn.erl:140: :elixir_-
fn.escape/3    (stdlib 4.2) lists.erl:1462: :lists.mapfoldl_-
1/3    (elixir 1.14.3) src/elixir_fn.erl:140: :elixir_-
fn.escape/3    (elixir 1.14.3) expanding macro: Kernel.
|>/2 # Encode function plaintext = "HELLO" encoded = plaintext
```

```
|> String.downcase() |> String.graphemes() |> Enum.map(
fn x ->   c = hd(String.to_charlist(x))   c = c - 97 -
11   c = rem(c, 26)   c = c + 97   c = Integer.to_string(
c)   <<c::utf8>> end) |> List.to_string() IO.puts(encoded)
 # Decode function ciphertext = "fthvq" decoded = ciphertext
|> String.graphemes() |> Enum.map(fn x ->   c = String.
to_integer(x |> then(fn x -> <<x::utf8>> end))   c = c
- 97 + 11   c = rem(c, 26)   c = c + 97   c = Integer.to_-
string(c)   <<c::utf8>> end) |> List.to_string() IO.puts(
decoded) # Encode function plaintext = "HELLO" encoded
= plaintext |> String.downcase() |> String.graphemes() |>
Enum.map(fn x ->   hd(String.to_charlist(x)) - 97 - 11
 |> rem(26)   |> & &1 + 97   |> Integer.to_string()   |>
List.to_string()   |> String.codepoints()   |> Enum.filter(
&String.printable?/1)   |> List.to_string() end) |> List.
to_string() IO.puts(encoded) # Decode function ciphertext
= "fthvq" decoded = ciphertext |> String.graphemes() |>
Enum.map(fn x ->   x   |> String.codepoints()   |> List.
to_string()   |> String.to_integer()   |> & &1 - 97 + 11
  |> rem(26)   |> & &1 + 97   |> Integer.to_string()
|> List.to_string()   |> String.codepoints()   |> Enum.
filter(&String.printable?/1)   |> List.to_string() end)
|> List.to_string() IO.puts(decoded) plaintext = "HELLO"
encoded = plaintext |> String.downcase() |> String.graphemes(
) |> Enum.map(fn x ->   %{value: char} = List.first(String.
codepoints(x))   char = char - ?a + 11 |> rem(26) |> Kernel.
+(?a)   Integer.to_string(char) end) |> List.to_string(
) |> String.replace("[]", "") |> IO.puts() ciphertext =
"fthvq" decoded = ciphertext |> String.graphemes() |> Enum.
map(fn x ->   %{value: char} = List.first(String.codepoints(
x))   char = char - ?a |> rem(26) |> Kernel.+(?a)   Integer.
to_string(char) end) |> List.to_string() |> String.replace(
"[]", "") |> IO.puts() shift_poly = [1, 0, 1] modular_-
inverse = fn f ->   fn a, n ->     case rem(n, a) do
   0 -> {1, 0, a}       b ->         {y, x, d} = f.(b,
a)         {x - div(n, a) * y, y, d}     end   end end.(
fn f -> &(&1.(&1.(&1))) end) poly_shift_character = fn
{char, _rest}, {a, b, c} ->   shifted_val = a * char *
char + b * char + c   <<shifted_val::utf8>> end encode
= fn f ->   fn [char | tail], key_poly ->     [poly_shift_-
```

```
character.({char, 0}, key_poly) | f.(tail, key_poly)]
 end.([], &1) end.(fn f -> &(&1.(&1.(&1))) end) decode
= fn f ->  fn [a, b | tail], {a_coeff, b_coeff, c_coeff}
->    inv_a = modular_inverse.(a_coeff, 256) |> elem(0)
    _inv_b = 256 - b_coeff    sqrt_term = round(:math.
sqrt(b_coeff * b_coeff - 4 * a_coeff * c_coeff))    inv_-
c = rem(inv_a * (b_coeff * b_coeff - 4 * a_coeff * c_coeff)
, 256)    char_val = rem(inv_a * (256 + b_coeff - sqrt_-
term), 256)                  |> rem(&(&1 in 32..126))
   [<<char_val::utf8>> | f.(tail, {a_coeff, b_coeff, c_-
coeff})]  end.([], &1) end.(fn f -> &(&1.(&1.(&1))) end)
 plaintext = "Hello World!" key_poly = shift_poly ciphertext
= encode.(String.graphemes(plaintext), key_poly) |> List.
to_string() decoded = decode.(String.codepoints(ciphertext)
, key_poly) |> List.to_string() # Output the results IO.
puts("Plaintext: #{plaintext}") IO.puts("Ciphertext: #{ciphertext}")
 IO.puts("Decoded: #{decoded}") defmodule Cipher do  def
shift_character(<<char::utf8>> = _input, shift) do
shifted_val = char + shift    <<shifted_val :: utf8>>
 end  def encode(plaintext, key) do    plaintext
|> String.codepoints()    |> Enum.map(&shift_character(
&1, key))    |> List.to_string()  end  def decode(ciphertext,
key) do    ciphertext    |> String.codepoints()    |>
Enum.map(&shift_character(&1, -key))    |> List.to_string(
)  end end plaintext = "Hello World!" k = 3 ciphertext
= Cipher.encode(plaintext, k) decoded = Cipher.decode(ciphertext,
k) IO.puts("Plaintext: #{plaintext}") IO.puts("Ciphertext:
 #{ciphertext}") IO.puts("Decoded: #{decoded}") defmodule
ShiftCipher do  defp modular_inverse(_a, 0), do: {1, 0,
0}  defp modular_inverse(a, n) when rem(n, a) != 0 do
  {y, x, d} = modular_inverse(rem(n, a), a)    {x - div(
n, a) * y, y, d}  end  defp poly_shift_character(char,
{a, b, c}) do    a * String.to_integer(char) * String.
to_integer(char) + b * String.to_integer(char) + c |> rem(
256)  end  def encode(plaintext, key_poly) do    plaintext
   |> String.codepoints()    |> Enum.map(&poly_shift_-
character(&1, key_poly))  end  def decode(ciphertext,
key_poly) do    ciphertext    |> String.codepoints()
  |> Enum.chunk_every(2, 1, :discard)    |> Enum.map(
&decode_character(&1, key_poly))    |> (fn x -> List.to_-
```

```
string(x) end).()  end  defp decode_character([a, b],
{a_coeff, b_coeff, c_coeff}) do    inv_a = modular_inverse(
a_coeff, 256) |> elem(0)    discriminant = b_coeff * b_-
coeff - 4 * a_coeff * c_coeff    case discriminant do
    d when d < 0 -> 0       _ ->         root1 = (-b_-
coeff + :math.sqrt(discriminant)) * inv_a |> rem(256)
      root2 = (-b_coeff - :math.sqrt(discriminant)) *
inv_a |> rem(256)         decoded =           case rem(
a, 2) do            0 ->              if rem(a * root1
* root1 + b * root1 + c_coeff, 256) == a do
    root1               else                root2
         end             _ ->               if b >= 0
do                 root1               else
   root2              end          end         Integer.
to_charlist(decoded)    end  end end plaintext = "Hello
World!" key_poly = {1, 0, 1} ciphertext = ShiftCipher.encode(
plaintext, key_poly) decoded = ShiftCipher.decode(ciphertext,
key_poly) IO.puts("Plaintext: #{plaintext}") IO.puts("Ciphertext:
 #{ciphertext}") IO.puts("Decoded: #{decoded}")
```