Algorithmen in der Kryptographie - Modellierung mit *DrScheme* und *DMdA* -

Gernot Lorenz

Version 1.2, September 2009

Algorithmen der Kryptographie bieten eine reichhaltiges Übungsfeld für das *funktionale* Modellieren und Programmieren, insbesondere für Datenmodellierung mittels Listen und den Umgang mit Prozeduren höherer Ordnung (Higher-Order-Programmierung). Die Umsetzung erfolgt mit dem Entwicklungssystem *DrScheme*[1] in der Lernsprachenversion *DMdA*[1]. Entsprechend werden beim Lernenden (Schüler oder Student) Grundkenntnisse vorausgesetzt.
Untersucht werden klassische Verfahren wie Substitution und Transposition.

Inhaltsverzeichnis

[1] Programmierumgebung für Anfänger *http://www.plt.org*

1 Einführung

1.1 Begrifflichkeit

Die *Kryptographie* beschäftigt sich mit der *Verschlüsselung (Chiffrierung)* eines *Klartextes (KT)* in einen *Geheimtext (GT)* oder allgemein *Chiffrat* sowie deren Umkehrung, also mit der *Entschlüsselung (Dechiffrierung)* eines *Geheimtextes* in einen *Klartext*:

$$\textit{Klartext} \xrightarrow[\text{Schlüssel}]{\text{Verschlüsseln (Chiffrieren)}} \textit{Geheimtext} \qquad \textit{Geheimtext} \xrightarrow[\text{Schlüssel}]{\text{Entschlüsseln (Dechiffrieren)}} \textit{Klartext}$$

Abb. 1: Kryptographie

Die *Kryptoanalyse* dagegen beschäftigt sich mit Methoden, aus einem Geheimtext den Klartext ohne Kenntnis des Schlüssels oder des Verfahrens oder beidem zu ermitteln.

Die historisch bekanntesten kryptographischen Verfahren beruhen auf

Substitution d.h. jeder Buchstabe des Klartextes wird durch einen (i.a. anderen) Buchstaben ersetzt. Entscheidend ist dabei der *Schlüssel*, in diesem Fall die Zuordnung zwischen *Klaralphabet (KA)* und *Geheimalphabet (GA)*. Allgemein gesprochen wird das KA durch eine Permutation zum GA. Beispiel:

```
KA: ABCDEFGHIJKLMNOPQRSTUVWXYZ
GA: NGPSBVWLHYKDIXMJFACRTUOZEQ
```

Mit diesem Schlüssel ergibt sich

```
KT: INFORMATIK
GT: HXVMAIRHK
```

Transposition Bei diesen Verfahren werden die Buchstaben des Klartextes nicht ersetzt, sondern permutiert (nicht zu Verwechseln mit der Permutation des KA bei Substitutionen!), d.h. sie tauchen an anderen Stellen auf, so dass insgesamt die Häufigkeit eines Buchstabens in Klartext und Geheimtext gleich ist. Beispiel:

```
KT: INFORMATIK
GT: RFOTIAMIKN
```

Zur Entschlüsselung benötigt man bei beiden Verfahren den Schlüssel und das Umkehrverfahren. Da bei Verschlüsselung und Entschlüsselung jeweils der gleiche Schlüssel benutzt wird, spricht man bei Substitution und Transposition von *symmetrischen* Verfahren.

1.2 Vereinbarungen

Da überlicherweise nur Großbuchstaben chiffriert und dechiffriert werden, sollen die zu konstruierenden Funktionen folgendes berücksichtigen:

- der Geheimtext sowie das Klaralphabet und das Geheimalphabet bestehen nur aus Großbuchstaben ($65 \leq \text{ASCII} \leq 90$)
- der Klartext kann aus Bequemlichkeitsgründen auch Kleinbuchstaben enthalten, also insgesamt Zeichen mit $65 \leq \text{ASCII} \leq 90$ und $97 \leq \text{ASCII} \leq 122$, versehentlich eingegebene Sonderzeichen werden ignoriert

1.3 Codierung

1.3.1 Allgemein

Die folgende Codierung ist mit der *DrScheme*-Version 4.1.5 im *DMdA*-Sprachlevel *Die Macht der Abstraktion mit Zuweisungen* entstanden; das bedeutet u.a., dass Testfälle als Bestandteil des Quellcodes mit `(check-expect <Funktionsaufruf> <Rückgabewert>))` auch hier aufgeführt werden.

1.3.2 DMdA vs. HtDP

Auch wenn Grundkenntnisse in *DrScheme* in den *HtDP*-Sprachen[2] vorausgesetzt werden, sind bei *DMdA*[1] folgende erhebliche Unterschiede zu beachten:

1. Funktionsverträge sind in *DMdA* Bestandteile des Quellcodes und haben die Form
   `(: <Bezeichner> (<Argument> ... -> <Rückgabewert>))`
   Zusätzlich werden sie im Folgenden in der *HtDP*-Form
   ; funktionsname: parmeterliste –> funktionswert
   mit sinntragenden Bezeichnern für die Paramter angegeben
2. Der Datentyp `char` ist nicht implementiert, entsprechend fehlen auch Umwandlungs-Funktionen wie z.B. `char->integer, integer->char` und `char-upcase`. Somit steht auch `string->list` nicht zur Verfügung, so dass eine Listenverarbeitung der ASCII-Nummern der Zeichen nicht möglich ist.
   Als Abhilfe werden hier die Funktionen
   `string->integer-list: string -> list of integer`, die eine Liste der ASCII-Nummern liefert, sowie die Umkehrfunktion
   `integer-list-> string: list of integer -> string`
   bereitgestellt in dem Teachpack `cryptography`[2]
3. Eine Funktionsdefinition hat im Gegensatz zu *HtDP* die Form

```
(define <Bezeichner>
   (lambda <Parameterliste>
      <Rumpf>))
```

4. Die Wahrheitswerte `true` und `false` lauten in *DMdA* `#t` und `#f`, also wie in Standard-Scheme
5. Die Higher-Order-Funktion `filter` ist nicht implementiert (im Gegensatz zu `map` und `apply`). Ebenso fehlt `member`. Sie werden im Folgenden definiert (s.u.)
   Ebenso fehlt
6. Die BOOLEsche Funktion *; member: any (listof any)- -> boolean* ist nicht implementiert. Sie wird im Folgenden definiert (s.u.)
7. Die Prüfung auf Gleichheit mit `equal?` ist erst *DMdA*-Stufe *Die Macht der Abstraktion mit Zuweisungen* möglich. Für *Die Macht der Abstraktion - Anfänger* und *Die Macht der Abstraktion* wird statt dessen das gewöhnliche Gleichheitszeichen = benutzt.

1.3.3 Hilfsfunktionen

Die in ihrer Bedeutung als bekannt vorausgesetzten Funktionen `filter` und `member` (s.o.) müssen selbst definiert werden:

```
; Vertrag
(: filter (%a (list %b) -> (list %b)))

;Testfall
(check-expect
 (filter even? (list 3 4 5))
 (list 4))

;Definition
```

[2] auf *http://web.me.com/lorenz07/MUPIS/* zu finden

```
(define filter
  (lambda
      (p? liste)
    (cond
      ((empty? liste) empty)
      ((p? (first liste))
       (cons (first liste) (filter p? (rest liste))))
      (else
       (filter p? (rest liste))))))
```

Die Funktion `member` wird mit `mitglied?` bezeichnet:

```
;Vertrag
(: mitglied? (\%a (list \%a) -> boolean))

;Testfall
(check-expect
 (mitglied? 3 (list 1 2 4 5))
 #f)

;Definition
(define mitglied?
  (lambda (element liste)
    (cond
      ((empty? liste) #f)
      ((equal? element (first liste)) #t)
      (else
       (mitglied? element (rest liste))))))
```

Wie oben bereits erwähnt, soll der Geheimtext nur aus Großbuchstaben bestehen, der Klartext dagegen darf aus Gründen der Bequemlichkeit auch Kleinbuchstaben enthalten, d.h. alle Sonderzeichen einschließlich dem Leerzeichen müssen vorher entfernt werden. Dies soll

```
; bereinige-klartext: zeichenkette --> KT
(: bereinige-klartext (string -> string))
```

leisten, und man erwartet z.B.

```
(check-expect
 (bereinige-klartext "Ad& am")
 "ADAM")
```

mit dem Gerüst

```
(define bereinige-klartext
  (lambda (klartext)
    ..... ))
```

Der als Zeichenkette einzugebende Klartext wird mittels `(string->integer-list ...)` in eine Zahlenliste (ASCII) umgewandelt, daraus die Nummern der Nicht-Buchstaben herausgefiltert, in der so entstandenen Liste die Nummern der Kleibuchstaben in die Nummern der zugehörigen Großbuchstaben umgewandelt und abschließend wieder mit `(integer-list->string ...)` in eine Zeichenkette, den Klartext KT umgewandelt:

```
(define bereinige-klartext
  (lambda (klartext)
    (integer-list->string
     (<Wandele evtl. Kleinbuchstaben in Großbuchstaben um>
      (<Filtere Sonderzeichen aus>
      (string->integer-list klartext))))))
```

Zunächst benötigen wir die Funktion

```
; nur-buchstaben: ASCII --> BOOLEsch
(: nur-buchstaben (natural -> boolean))
```

mit z.B.

```
(check-expect
 (nur-buchstaben 28)
 #f)

(check-expect
 (nur-buchstaben 118)
 #t)
```

mit

```
(lambda (ASCII)
         (cond
           ((or (<= 65 ASCII 90) (<= 97 ASCII 122)) #t)
           (else #f)))
```

und erhalten so durch den Aufruf

```
(filter nur-buchstaben (string->integer-list "Ad& am")) --> (list 65 100 97 109)
```

Darin sind drei Nummern von Kleinbuchstaben enthalten, nämlich 100, 97 und 109, die mittels

```
; GROSS-buchstaben: ASCII --> ASCII
(: GROSS-buchstaben (natural -> natural))
```

mit z.B.

```
(check-expect
 (GROSS-buchstaben 28)
 28)

(check-expect
 (GROSS-buchstaben 118)
 86)
```

und und Quellcode

```
(define GROSS-buchstaben
(lambda (znr)
        (cond
          ((<= 97 znr 122) (- znr 32))
          (else znr))))
```

die mittels `map` auf die o.a. Zahlenliste angewendet wird, also z.B.

```
(map GROSS-buchstaben (list 65 100 97 109)) --> (list 65 68 65 77)
```

Damit können wir obige Schablone komplettieren:

```
(define bereinige-klartext
  (lambda (klartext)
    (integer-list->string
     (map
      GROSS-buchstaben
      (filter
       nur-buchstaben
       (string->integer-list klartext))))))
```

2 Monoalphabetische Substitution

2.1 Allgemeines Verfahren

Bei allen monoalphabetischen Substitutionen, den sog. *Cäsar*-Verschlüsselungen, ist das Verfahren gleich: jeder Buchstabe des KT wird anhand der Zuordnung KA → GA ersetzt (substituiert), wobei das GA eine bestimmte oder zufällig erzeugte Permutation des KA ist (im Gegensatz dazu werden bei polyalphabetischen Substitutionen mehrere GAs benutzt).

Beispiel:

```
KA: ABCDEFGHIJKLMNOPQRSTUVWXYZ
GA: GQFALNDZJXPIKMEHWBOTVSRCYU
```

Für den Klartext „ADAM" bedeutet das: Das A wird durch G, das D durch A, das A wieder durch G und das M durch K ersetzt, wodurch sich der Geheimtext „GAGK" ergibt:

```
KT: ADAM
GT: GAGK
```

Bei Entschlüsselung bzw. Dechiffrierung ist offensichtlich umgekehrt zu substituieren, d.h. KA und GA sind zu vertauschen, aber der Substituions-Vorgang S bleibt der gleiche, unabhängig davon, welches der nachfolgenden Verfahren („klassischer" Cäsar (Verschiebe-Cäsar), Zufalls-Cäsar (zufällige Permutation von KA) oder Cäsar mit Losungswort) man betrachtet. Somit unterscheiden sich diese Verfahren nur durch die Art, wie das KA gewonnen wird:

$$\text{KT} \xrightarrow{\text{S(KA,GA)}} \text{GT} \qquad \text{und} \qquad \text{GT} \xrightarrow{\text{S}^{-1}\text{(KA,GA)}} \text{KT}$$

Abb. 2: monoalphabetische Ver- und Entschlüsselung

wobei S^{-1}(KA,GA) = S(GA,KA), d.h. beim Entschlüsseln hat man das gleiche Verfahren, vertauscht aber KA und GA.

Es bleiben zwei Aufgaben:

1. Wie lautet die Funktion für monoalpahabetische Substitution?
2. Wie erhält man das GA für das jeweilige Verfahren?

Wir beginnen wir mit 1.: Die Funktion

```
; substituttion: KT KA GA --> GT
(: substitution (string string string -> string))
```

soll einen Klartext anhand des Klaralphabets KA und des Geheimalphabets GA in einen Text umwandeln. Dabei wird das Klaralphabet auch als Parameter eingesetzt, da es beim Dechiffrieren als Geheimalphabet benötigt wird, und setzen "ABCDEFGHIJKLMNOPQRSTUVWXYZ" als „normales" KA:

```
(define standardKA "ABCDEFGHIJKLMNOPQRSTUVWXYZ")
```

Mit z.B.

```
(define einGA "GQFALNDZJXPIKMEHWBOTVSRCYU")
```

erwarten wir

```
(check-expect
 (substitution "ADAM" standardKA einGA)
 "GAGK")
(check-expect
 (substitution "A&!D aM" standardKA einGA)
 "GAGK")
```

und für die Entschlüsselung

```
(check-expect
 (substitution "GAGK" einGA standardKA)
 "ADAM")
(check-expect
 (substitution (substitution "ADAM" standardKA einGA) einGA standardKA )
 "ADAM")
```

Der Klartext wird, nachdem er von Sonderzeichen mittels `bereinige-klartext` befreit worden ist, mittels `string->integer-list` in eine Liste von Zahlen, die die ASCII-Nummern darstellen, umgewandelt, dann erfolgt die eigentliche Substitution, und anschließend wird so erhaltene Zahlenliste wieder in eine Zeichenkette, den GT, transformiert. Das ergibt die Schablone:

```
(define substitution
  (lambda (klartext einKA einGA)
    (integer-list->string
        ...<Substitution mittels KA und GA>...
      (string->integer-list (bereinige-klartext klartext))))))
```

Die eigentliche Substitution erfolgt zeichenweise: Um in unserem Beispiel das `"A"` von `"ADAM"` zu `"G"` zu verschlüsseln, muss in der Liste

```
(string->integer-list (bereinige-klartext "ADAM")) --> (list 65 68 65 77)
```

die Position von 65 gefunden werden, nämlich 0, und entsprechend das 0. Element in der GA-Liste

```
(string->integer-list "GQFALNDZJXPIKMEHWBOTVSRCYU")
--> (list 71 81 70 65 76 78 68 90 74 88 80 73 75 77 69 72 87 66 79 84 86 83 82 67 89 85)
```

mitels `list-ref` , was 71 liefert.
Allgemein gesagt, wird für jede ASCII-Nr. mittels `stelle` die Position in dem ebenfalls in eine Zahlenliste umgewandelten Klaralphabet ermittelt und mittels `list-ref` die entsprechende Zahl, also die neue ASCII-Nr. an der gleichen Stelle im Geheimalphabet (das ebenfalls in eine Zahlenliste umgewandelt wird) festgestellt; wir erhalten als Term:

```
        (list-ref (string->integer-list GA)
                  (stelle ASCII-Nr (string->integer-list KA))))
```

Diesen Term können wir als anonyme Funktion mit einem Parameter

```
(lambda (ASCII-Nr)
        (list-ref (string->integer-list GA)
                  (stelle ASCII-Nr (string->integer-list KA))))
```

mit Hilfe von `map` auf den KT als Zahlenliste anwenden.
Während die Funktion *; list-ref: liste element-nr –> element* in PLT-Scheme eingebaut ist, müssen wir noch die Funktion `stelle` definieren, mit der wir herauszufinden können, an welcher Stelle $0, 1, ..., n-1$ in einer n-elementigen Zahlenliste eine bestimmte Zahl steht:

```
; stelle: zahl liste --> zahl
(: stelle (number (list number) -> number))
```

definiert, und man erwartet z.B.

```
(check-expect
 (stelle 7 (list  4 5 6 7 8))
 3)
```

und erhalten als Gerüst

```
(define stelle
  (lambda (zahl liste)
    .....))
```

Da die vorgegebene Zahl nacheinander mit den Elementen der Zahlenliste verglichen werden muss und somit eine Liste durchlaufen wird, liegt der Fall eines gemischten Datentyps vor und die Funktionsschablone muss aus einer Verzweigung bestehen:

```
(define stelle
  (lambda (zahl liste)
    (cond
      (... ...)
      (... ...)
```

Genau genau genommen liegen sogar drei Fälle: Neben dem Fall der leeren Liste kann es sein, dass die vorgegebne Zahl mit dem ersten Element der Liste bereits übereinstimmt, wofür wir den Wert 0 erwarten, oder aber nicht:

```
(define stelle
  (lambda (zahl liste)
    (cond
      ((empty? liste) ...)
      ((= zahl (first liste)) 0)
      (else
       (.... (stelle zahl (rest liste) ...))))))
```

Da sich bei jedem Selbstaufruf die gesuchte Stelle um den Wert 1 erhöht, ergibt sich

```
(define stelle
  (lambda (zahl liste)
    (cond
      ((empty? liste) ...)
      ((= zahl (first liste)) 0)
      (else
       (+ 1 (stelle zahl (rest liste)))))))
```

Es bleiben zwei Sonderfälle, die geklärt werden müssen, auch wenn sie in der beabsichtigten Anwendung nicht eintreten werden: Im Falle einer leeren Liste können wir gefahrlos den Wert -1 setzen, aber in der bisherigen Schablone ist der Fall, dass die Zahl in der Liste nicht enthalten ist, nicht vorgesehen: Diesen Fall können wir mit der Funktion `(mitglied?` abprüfen und im positiven Fall ebenfalls mit -1 bewerten:

```
(check-expect
 (stelle 3 (list  4 5 6 7 8))
 -1)
(check-expect
 (stelle 3 empty)
 -1)
```

Insgesamt erhalten wir:

```
(define stelle
  (lambda (zahl liste)
    (cond
      ((empty? liste) -1)
      ((not (mitglied? zahl liste)) -1)
      ((= zahl (first liste)) 0)
      (else
       (+ 1 (stelle zahl (rest liste)))))))
```

Insgesamt ergibt sich:

```
(define substitution
  (lambda (klartext KA GA)
    (integer-list->string
     (map
      (lambda (ASCII-Nr)
        (list-ref (string->integer-list GA)
                  (stelle ASCII-Nr (string->integer-list KA))))
      (string->integer-list (bereinige-klartext klartext))))))
```

Es bleibt die Frage nach der Gewinnung des GA, der im folgenden für die verschiedenen Substitutionsverfahren nachgegangen wird.

2.2 Alphabetischer- oder Verschiebe-Cäsar

Bei diesem „klassischen" Cäsar ist das GA um k Stellen gegenüber dem KA zyklisch verschoben. Für z.B. $k = 3$ bedeutet das, dass das GA mit dem Buchtaben D beginnt, die fehlenden Buchstaben A, B und C werden am Ende aufgefüllt:

```
KA: ABCDEFGHIJKLMNOPQRSTUVWXYZ
GA: DEFGHIJKLMNOPQRSTUVWXYZABC
```

Für eine mögliche Funktion zur Gewinnung von GA aus KA ist eine Funktion

```
; GA-macher-verschiebe: verschiebung KA --> GA
(: GA-macher-verschiebe (natural string -> string))
;erzeugt aus KA ein zyklisch um eine Zahl verschiebung verschobenes GA
```

schnell gefunden.
Wir erwarten z.B. folgende Aufrufe und Ergbnisse:

```
(check-expect
 (GA-macher-verschiebe 3 standardKA)
 "DEFGHIJKLMNOPQRSTUVWXYZABC")
```

und insgesamt

```
(check-expect
 (substitution "ADAM" standardKA
        (GA-macher-verschiebe 3 standardKA))
 "DGDP")
(check-expect
 (substitution "Das ist ein Geheimtext" standardKA
        (GA-macher-verschiebe 3 standardKA))
 "GDVLVWHLQJHKHLPWHAW")
(check-expect
 (substitution "Dasist§..ei34nGehe ?imtext" standardKA
        (GA-macher-verschiebe 3 standardKA))
 "GDVLVWHLQJHKHLPWHAW")
```

Diese Prozedur entspricht auch einem Substitutionsverfahren: von der ASCII-Nr. eines Buchstabens wird 65 subtrahiert und zu der Differenz die Verschiebung addiert. Das Ergebnis wird mit Rest durch 26 geteilt, damit bei Überschreitung von 26 der Rest die richtige ASCII-Nr. angibt. Zu dieser wird wieder 65 addiert.
Der zugehörige Term

```
(+ (modulo (+ (- ASCII-Nr 65) verschiebung) 26) 65))
```

wird als anonyme Funktion mittels `map` mittels `string->integer-list` auf das in eine Zahlenliste umgewandelte KA angewendet und das Ergebnis mittels `integer-list->string` zum Funktionswert gemacht:

```
(define GA-macher-verschiebe
  (lambda (verschiebung KA)
      (integer-list->string
       (map
        (lambda (ASCII-Nr)
          (+ (modulo (+ (- ASCII-Nr 65) verschiebung) 26) 65))
        (string->integer-list KA)))))
```

Alternative: verkürztes Verfahren

Genau genommen, wird bei diesem Verschiebe-Cäsar zweimal substituiert, sowohl bei `substitution` bei als auch bei `GA-macher-verschiebe` .
Wann man also an keiner weiteren monoalphabetischen Substitution interessiert ist, kann man den

Aufwand reduzieren durch eine einmalige Substitution, in dem die Buchstaben des KT direkt modulo 26 verschoben werden, so dass man die Hilfsfunktion `GA-macher-verschiebe` nicht benötigt.

Dazu führt man die Verschiebung als zweiten Parameter ein:

```
; caesar-verschiebe: KT verschiebezahl --> GT
(: caesar-verschiebe (string natural -> string))
```

mit z.B.

```
(check-expect
 (caesar-verschiebe "ADAM" 3)
 "DGDP")
```

und erhält

```
;erzeugt GT aus KT direkt (ohne GA-Erzeugung)
(define caesar-verschiebe
  (lambda (klartext verschiebung)
    (integer-list->string
     (map
      (lambda (ASCII-Nr)
        (+ (modulo
            (+ (- ASCII-Nr 65) verschiebung) 26) 65))
      (string->integer-list (bereinige-klartext klartext))))))
```

Entschlüsselung

Grundsätzlich wird bei symmetrischen Verfahren beim Entschlüsseln der gleiche Schlüssel wie beim Verschlüsseln benutzt, allerdings mit anderem Algorithmus. Dagegen ist es bei monoalphabetischen Substitutionen auch möglich – wenn man, wie oben geschehen, das Verfahren aufteilt in Gewinnung des GA und die eigentliche `substitution` – das Entschlüsseln durch Vertauschung von KA und GA zu erreichen:

```
(check-expect
 (substitution "DGDP" (GA-macher-verschiebe 3 standardKA) standardKA)
 "ADAM")
(check-expect
 (substitution
  (substitution "ADAM" standardKA (GA-macher-verschiebe 3 standardKA))
  (GA-macher-verschiebe 3 standardKA) standardKA)
 "ADAM")
```

Bei dem abgekürzten Verfahren `caesar-verschiebe` ist eine solche Vertauschung natürlich nicht möglich.
Stattdessen kann man eine eigenständige Funktion

```
; de-caesar-verschiebe: GT verschiebung --> KT
(: de-caesar-verschiebe (string natural -> string))
```

definieren mit z.B.

```
(check-expect (de-caesar-verschiebe "DGDP" 3) "ADAM")
```

wobei man die für die Entschlüsselung notwendige „Gegenschiebung" $26 - k$ erreicht, indem man `caesar-verschiebe` mit $26 - k$ als Wert für die Verschiebung aufruft:

```
(define de-caesar-verschiebe
  (lambda (geheimtext verschiebung)
    (caesar-verschiebe geheimtext (- 26 verschiebung))))
```

2.3 Zufällige Permutation von KA: Buchstaben-Cäsar

Hierbei erhält man das GA durch beliebige (zufällige) Permutation von KA, etwa

```
GA: GMAXYVFLNZBUQJWIOTDEKPCSRH
```

mit

```
(check-expect
 (substitution "ADAM" standardKA "GMAXYVFLNZBUQJWIOTDEKPCSRH")
 "GXGQ")
```

Um ein solches zufälliges Geheimalphabet zu erzeugen, braucht man eine Funktion

```
; GA-macher-zufall: KA --> GA
(: GA-macher-zufall (string -> string))
```

die das Standard-KA permutiert.
Da es dafür bekanntlich 26! = 403291461126605635584000000 Möglichkeiten gibt, sind hier Testfälle mit `check-expect` begrenzt hilfreich:

```
(check-expect
 (GA-macher-zufall2 standardKA)
 "TECGISYMFVXKROANWZBUQDLPJH")
```

Das eigentliche Permutieren kann dadurch erreicht werden, dass das KA als Liste von Einzelzeichen durchlaufen wird und dabei die einzelnen Zeichen in eine (zunächst leere) Liste an zufälliger Stelle eingefügt werden. Dieser Vorgang soll durch die Hilfsfunktion
`;GA-macher-zufall-hilfe: KA-Liste -> GA-Liste` umgesetzt werden:

```
(define GA-macher-zufall
  (lambda (KA)
      (integer-list->string
       (GA-macher-zufall-hilfe (string->integer-list KA)))))
```

Ein Gerüst für die Hilfsfunktion

```
; GA-macher-zufall-hilfe: KA-Liste Liste(leer) --> GA-Liste
(: GA-macher-zufall-hilfe ((list %a) (list %b) -> (list %c)))
```

mit z.B.

```
(check-expect
 (GA-macher-zufall-hilfe (list "A" "B" "C" "D" "E") (list "X" "Y" "Z"))
 ((list "A" "X" "B" "C" "D" "Y" "Z" "E"))
(check-expect
 (GA-macher-zufall-hilfe (list "A" "B" "C" "D" "E") empty)
 (list "E" "B" "A" "C" "D"))
```

ist folgendes:

```
(define GA-macher-zufall-hilfe
  (lambda (KA-liste GA-Liste)
    ....))
```

Da eine Liste durchlaufen wird, ergibt sich als Schablone:

```
(define GA-macher-zufall-hilfe
  (lambda (KA-liste GA-Liste)
    (cond
      ((empty? KA-liste) .....)
      (else
        (GA-macher-zufall-hilfe (rest KA-Liste) .....))))))
```

Die erste Ellipse ist leicht gefüllt: Ist die einzufügende Liste leer, wird die unveränderte zweite Liste, also `GA-Liste` zurückgegeben.
Die zweite Ellipse steht im Selbstaufruf für die aufzufüllende Liste, die dadurch gebildet wird, dass jeweils das erste Element der KA-Liste in die GA-Liste an einer zufälligen Position eingefügt wird, was durch eine Funktion

```
; in-liste-einfuegen: element liste stelle --> liste
(: in-liste-einfuegen (%a (list %b) number -> (list %b)))
```

erreicht werden kann, mit z.B.

```
(check-expect
 (in-liste-einfuegen "A" (list "X" "Y" "Z") 2)
 (list "X" "Y" "A" "Z"))
```

Wir haben somit als Gerüst:

```
(define in-liste-einfuegen
  (lambda (element liste wo)
    .....))
```

Da eine Liste durchlaufen wird, ergibt sich

```
(define in-liste-einfuegen
  (lambda (element liste wo)
    (cond
      ((empty? liste) .....)
      (.....))))
```

Wenn die Liste nicht leer ist, ergeben sich zwei Fälle: Das Element soll am Anfang der Liste eingefügt werden, d.h. die Position und damit der Wert von `wo` ist gleich Null, oder an einer späteren Stelle:

```
(define in-liste-einfuegen
  (lambda (element liste wo)
    (cond
      ((empty? liste) .....)
      ((zero? wo) .....)
      (else
       .....))))
```

Bei einer leeren Liste entsteht eine neue mit dem einem Element, nämlich `(list element` . Im Fall `(zero? wo)` ergibt sich einfacherweise `(cons element liste)` .
Im dritten Fall, d.h. wenn $wo > 1$, besteht die neue Liste auf jeden aus dem ersten Element der bisherigen Liste und einem Rest, den man dadurch erhält, dass die Funktion auf der bisherigen Liste anwendet, allerdings an der Stelle `(- wo 1)` , da ja mit jedem Selbstaufruf die relative Position um 1 vermindert wird:

```
(define in-liste-einfuegen
  (lambda (element liste wo)
    (cond
      ((empty? liste) (list element))
      ((zero? wo) (cons element liste))
      (else
       (cons (first liste) (in-liste-einfuegen element (rest liste) (- wo 1)))))))
```

Wir kehren zurück zu unserer Hilfsfunktion

```
(define GA-macher-zufall-hilfe
  (lambda (KA-liste GA-liste)
    (cond
      ((empty? KA-liste) GA-liste)
      (else
       (GA-macher-zufall-hilfe
        (rest KA-liste)
        (in-liste-einfuegen   .....    .....   .....)))))))
```

und müssen noch klären, mit welchen Parameterwerten `in-liste-einfuegen` aufgerufen wird. Für das einzufügende Element, also die erste Ellipse ergibt sich `(first KA-liste)`, da ja die KA-Liste durchlaufen wird. Die zweite Ellipse stellt die Zielliste dar, also `GA-Liste`. Da die Einfügeposition zufällig sein soll, muss sie mittels $0 \leq$ `(random n)` $< n - 1$ bestimmt werden: Eine Liste mit n Elementen bietet $n + 1$ Einfügepositionen, also lautet die dritte Ellipse `(random (+ 1 (length GA-liste))`.
Insgesamt ergibt sich:

```
(define GA-macher-zufall-hilfe
  (lambda (KA-liste GA-liste)
    (cond
      ((empty? KA-liste) GA-liste)
      (else
       (GA-macher-zufall-hilfe
        (rest KA-liste)
        (in-liste-einfuegen
         (first KA-liste) GA-liste (random (+ 1 (length GA-liste)))))))))

(define GA-macher-zufall
  (lambda (KA)
      (integer-list->string
       (GA-macher-zufall-hilfe (string->integer-list KA)))))
```

Entschlüsselung

Die Dechiffrierung erfolgt wieder durch Vertauschung von KA und GA, wobei zu beachten ist, dass ein Testfall wie

```
(check-expect
 (substitution
  (substitution "ADAM" standardKA (GA-macher-zufall standardKA))
  (GA-macher-zufall standardKA) standardKA)
 "ADAM")
```

nicht sinnvoll ist, da `GA-macher-zufall` als Zufallsfunktion zweimal aufgerufen wird.

2.4 Substitution mit Losungswort (Schlüsselwort)

Bei diesem-Verfahren beginnt das Geheimalphabet mit einem leicht zu merkenden Wort (Länge $<$ 26), wobei Buchstaben, die sich im Losungswort wiederholen, im weiteren Verlauf entfallen. Der Rest wird mit den fehlenden Buchstaben aufgefüllt.
Beispiel: Mit „Dechiffrieren" als Losungswort ergibt sich:

```
KA: ABCDEFGHIJKLMNOPQRSTUVWXYZ
GA: DECHIFRNABGJKLMOPQSTUVWXYZ
```

Von der gesuchten Funktion

```
; GA-macher-losung: losungswort KA --> GA
(: GA-macher-losung (string string -> string))
;erzeugt aus Losungswort und KA ein GA
```

erwarten wir z.B.

```
(check-expect
 (GA-macher-losung "Dechiffrieren" standardKA)
 "DECHIFRNABGJKLMOPQSTUVWXYZ")
(check-expect
 (GA-macher-losung "LOSUNG" standardKA)
 "LOSUNGABCDEFHIJKMPQRTVWXYZ")
(check-expect
 (substitution
  (substitution "ADa §M" standardKA (GA-macher-losung "LOSUNG" standardKA))
  (GA-macher-losung "LOSUNG" standardKA) standardKA)
 "ADAM")
```

Der Quellcode von `GA-macher-losung` wird hier ohne Erläuterungen zum intensiven Selbststudium angeben: (s. Übungen)

```
(define GA-macher-losung
  (lambda (losungswort KA)
    (letrec
        ((stripper-hilfe
          (lambda (liste vorgekommen)
            (cond
              ((empty? liste) empty)
              ((mitglied? (first liste) vorgekommen)
               (stripper-hilfe (rest liste) (cons (first liste) vorgekommen)))
              (else
               (cons (first liste)
                     (stripper-hilfe
                      (rest liste) (cons (first liste) vorgekommen)))))))
         (stripper-rest
          (lambda (zeichenkette)
            (let
                ((GATeil1 (stripper-hilfe
                           (string->integer-list zeichenkette) empty)))
              (integer-list->string
               (filter
                (lambda (element)
                  (cond
                    ((mitglied? element GATeil1) #f)
                    (else
                     #t)))
                (string->integer-list KA))))))
         (stripper
          (lambda (zeichenkette)
            (string-append
             (integer-list->string
              (stripper-hilfe (string->integer-list zeichenkette) empty))
             (stripper-rest zeichenkette)))))
      (stripper (bereinige-klartext losungswort)))))
```

Noch ein Testlauf:

```
(check-expect
 (substitution "Es lohnt sich" standardKA
               (GA-macher-losung "SELBSTSTUDIUM" standardKA))
 "TPFJIHQPMLI")
```

Entschlüsselung

Die Dechiffrierung erfolgt wieder durch Vertauschung von KA und GA, z.B.

```
(check-expect
 (substitution
  (substitution "ADa $M" standardKA (GA-macher-losung "LOSUNG" standardKA))
      (GA-macher-losung "LOSUNG" standardKA) standardKA)
 "ADAM")
```

Übungen

1. Bearbeite *Substitution durch Losungswort* folgendermaßen:
   a) Analysiere den Quellcode ausführlich und versuche, das Verfahren und Funktion zu verstehen
   b) Definiere `stripper-hilfe, stripper-rest` und`stripper` als globale Funktionen und passe `GA-macher-losung` entsprechend an.

3 Polyalphabetische Substitution

3.1 Vigenére/Trithemius-Chiffre

Das auf Ideen von *Leon Battista Alberti*, *Johannes Trithemius* und *Giovanni Porta* beruhende und von *Blaise de Vigenère* entwickelte Verfahren benutzt 26 verschiedene Geheimalphabete zum Verschlüsseln, und zwar sämtliche Cäsar-Verschiebungen (s.o.).
Um nun festzulegen, welcher Buchstabe des Klartextes mit welchem Verschiebe-Cäsar verschlüsselt wird, ist ein Schlüsselwort notwendig, dazu ein Beispiel mit dem Schlüsselwort `INFORMATIK` und dem Klartext `Kryptographie macht Freude` :

```
Schlüssel:       INFORMATIKINFORMATIKINFO
KT, bereinigt:   KRYPTOGRAPHIEMACHTFREUDE
GT:              SEDDKAGKIZPVJAROHMNBMHIS
```

Ist das Schlüsselwort kürzer als der (bereinigte) KT, wird es so lange wiederholt, bis beide die gleiche Länge haben oder anders gesagt, bis jeder Buchstabe des KT mit einem Buchstaben des Schlüsselwortes verknüpft ist, so dass bei diesem Beispeil folgende Buchstabenpaare entstehen: (I, K), (N, R), (F, Y), (O, P), usw.
Der erste Buchstabe in einem Paar, also am Anfang beim ersten Paar das I, gibt das Geheimalphabet (das man durch Cäser-Verschiebung mit $k = 8$ mittels `(GA-macher-verschiebe 8) -> "IJKLMNOPQRSTUVWXYZABCDEFGH"` erhält) an, mit dem der zweite Buchstabe im Paar, also das K zu S chiffriert wird:

Schlüssel-Zeichen	Verschiebung k	GA	Chiffr.
I	8	ABCDEFGHIJ**K**LMNOPQRSTUVWXYZ **I**JKLMNOPQR**S**TUVWXYZABCDEFGH	K -> S
N	13	ABCDEFGHIJKLMNOPQ**R**STUVWXYZ **N**OPQRSTUVWXYZABCD**E**FGHIJKLM	R -> E
F	5	ABCDEFGHIJKLMNOPQRSTUVWX**Y**Z **F**GHIJKLMNOPQRSTUVWXYZABC**D**E	Y -> D
...	...	...	...

Für unsere gesuchte Funktion

```
; vigenere: KT schluesselwort --> GT
(: vigenere (string string -> string))
```

erwarten wir z.B.

```
(check-expect
   (vigenere "KRYPTOGRAPHIEMACHTFREUDE" "INFORMATIK")
   "SEDDKAGKIZPVJAROHMNBMHIS")
(check-expect
   (vigenere "ADAM" "EVA") "EYAQ")
(check-expect
   (vigenere "ADa $M" "EVA") "EYAQ")
```

Wenn für die einzelnen Cäsar-Verschiebung auch die bereits definierte Funktion `(substitution <klartext> <KA> <GA>)` zusammen mit `(GA-macher-verschiebe <verschiebung>)` verwendet werden soll, muss berücksichtigt werden, dass jetzt für jedes Zeichen des KT ein anderes GA benötigt wird, d.h. die Zeichenkette des Klartextes muss zerlegt werden in einzelne Zeichenketten der Länge 1.
Dies kann z.B. durch

```
(string->strings-list "ADAM") -->  (list "A" "D" "A" "M")
(string->strings-list "INFORMATIK")
        -->  (list "I" "N" "F" "O" "R" "M" "A" "T" "I" "K" )
```

geschehen, sowie die Umkehrung mittels

```
(strings-list-> string (list "A" "D" "A" "M") -->  "ADAM"
(strings-list-> string (list "I" "N" "F" "O" "R" "M" "A" "T" "I" "K" )
        -->  "INFORMATIK"
```

Als Gerüst ergibt sich somit

```
(define vigenere
  (lambda (klartext sw)
   ....
      (strings-list->string
       (vigenere-hilfe
        .... (string->strings-list (bereinige-klartext klartext)) ....))))
```

wobei die eigentliche Verschlüsselung durch die Hilfsfunktion

```
; vigenere-hilfe: KT-Liste schluesselwort laenge-KT-Liste --> GT-Liste
(: vigenere-hilfe ((list string) string natural -> (list string)))
```

erfolgt, wobei der zweite Parameter das Schlüsselwort repräsentiert und der dritte Parameter die Länge des (bereinigten) Klartextes ist, wie weiter unten begründet wird.
Der erste Parameter stellt den Klartext als Liste von Einzel-Strings dar, die von der Funktion durchlaufen wird. Dabei wird auf jeden Einzelstring die Funktion `(substitution <klartext> <KA> <GA>)` angewendet. Die jeweiligen Rückgabewerte sind wieder Strings der Länge 1 und werden mittels `cons` wieder zu einer Liste der Einzelzeichen des Geheimtextes zusammengefügt.
Man erwartet z.B.:

```
(check-expect
 (vigenere-hilfe (list "A" "D" "A" "M") "EVA" 4)
 (list "E" "Y" "A" "Q"))
```

Entsprechend hat die Schablone etwa folgende Form:

```
(define vigenere-hilfe
  (lambda (liste .....)
    ....
      (cond
        ((empty? liste) empty)
        (else
         (cons
           (substitution
            (first liste) standardKA (GA-macher-verschiebe <Verschiebung k>))
           (vigenere-hilfe  (rest liste) ..... )))))))
```

Diese Funktion wird in `vigenere` mit `(vigenere-hilfe klartext-liste .....)` aufgerufen, d.h. in obigem Beispiel mit dem Wert `(list "K" "R" "Y" "P" "T" "O" "G" "R" "A" "P" "H" "I" "E")` für den Parameter `liste`. Die folgenden Selbstaufrufe mit dem Rest der Liste haben zur Folge, dass das jeweils zu verschlüsselnde Zeichen das erste Element der Liste ist und sich die Länge der Liste jeweils um 1 vermindert.
Kernstück ist dabei die Ermittlung der Verschiebung k für jede Einzel-Substitution durch den Aufruf

```
(list-ref sw-liste (modulo (- laenge-KT (length liste)) (length sw-liste)))
```

Dabei sind

- `sw-liste` Schlüsselwort als Liste `(list "I" "N" "F" "O" "R" "M" "A" "T" "I" "K" )`
- `laenge-KT` die (feste) Länge des Klartextes, hier 24
- `(length liste)` die (veränderliche) Länge des Klartext-Rests

Für das erste Paar (I, K) in obigem Beispiel bedeutet das: Für das erste zu verschlüsselnde Zeichen `K` wird das erste Zeichen des Schlüssel benötigt, also das `I`. Am Anfang haben `laenge-KT` und `(length liste)` den gleichen Wert, nämlich 24, was die Differenz 0 ergibt und somit den Index 0 für `list-ref` , was den Wert `"I"` liefert. Allerdings wird für spätere Fälle der Rest `(modulo ..... laenge-sw` genommen, da sich das Schlüsselwort im Schlüssel wiederholt.
Um nun die Verschiebung k zu erhalten, wird von `"I"` mittels `(first (string->integer-list "I"))` die ASCII-Nr. ermittelt, von der aus bekannten Gründen 65 subtrahiert wird.

Ergänzt man die Parameterliste noch um die fehlenden Parameter `sw` (Schlüsselwort) sowie `laenge-KT` ((feste) Länge des Klartextes) und deklariert man `sw-liste` als lokale Variable, ist der Quellcode komplett (s.u.).

Im Hauptprogramm `vigenere` ist noch die lokale Definition

```
(let ((klartext-liste (string->strings-list (bereinige-klartext klartext))))
```

zu ergänzen, so dass dort der Aufruf `(vigenere-hilfe klartext-liste sw (length klartext-liste))` erfolgen kann. Es ergibt sich insgesamt:

```
(define vigenere-hilfe
  (lambda (liste sw laenge-KT)
    (let ((sw-liste (string->strings-list sw)))
      (cond
        ((empty? liste) empty)
        (else
         (cons
          (substitution
           (first liste)
           standardKA
           (GA-macher-verschiebe
            (- (first
                (string->integer-list
                 (list-ref
                  sw-liste (modulo (- laenge-KT (length liste)) (length sw-liste)))))
               65) standardKA))
          (vigenere-hilfe  (rest liste) sw laenge-KT)))))))
```

```
(define vigenere
  (lambda (klartext sw)
    (let ((klartext-liste (string->strings-list (bereinige-klartext klartext))))
      (strings-list->string
       (vigenere-hilfe
        klartext-liste sw (length klartext-liste))))))
```

Entschlüsselung

Die einfachste Möglichkeit zur Konstruktion einer Funktion

```
; de-vigenere: GT sw --> KT
(: de-vigenere (string string -> string))
```

liefert die Überlegung, dass bei jeder Einzel-Cäsarverschiebung beim Aufruf von `(GA-macher-verschiebe <k>)` in obigem Quellcode der Term zur Bestimmung der Verschiebung k durch die „Rückschiebung" $26 - k$ ersetzt wird, also `(GA-macher-verschiebe (- 26 <k>))`.

Eine andere Lösung stellt die Ermittlung eines „Dechiffrier-Schlüsselwortes" aus dem ursprünglichen Schlüsselwortes dar, und zwar mittels einer Hilfsfunktion

```
; de-sw: sw --> dechiffrier-sw
(: de-sw (string -> string))
```

dar, so dass z.B.

```
(check-expect
 (de-sw "EVA")
 "WFA")
(check-expect
```

```
 (vigenere (vigenere "ADAM" "EVA") "WFA")
 "ADAM")
(check-expect
 (vigenere (vigenere "ADAM" "EVA") (de-sw "EVA"))
 "ADAM")
```

zu erwarten ist und die Entschlüsselung mittels `(vigenere klartext (de-sw sw))` erfolgen kann.

Die gesuchte Funktion muss wieder die Zeichenkette des Schlüsselwortes in eine Zahlenliste der ASCII-Nr. umwandeln, auf der eine Funktion operieren muss, die die ASCII-Nr. der „Rückschiebe"-Buchstaben ermittelt, worauf die neue ASCII-Nr.-Liste wieder in eine Zeichenkette transformiert werden muss:

```
(define de-sw
  (lambda (sw)
    (integer-list->string
     (map
      (lambda (nr)
        < ... Ermittlung der "Rückschiebe"-ASCII-Nr. ... >
      (string->integer-list sw)))))
```

Will man vermeiden, dass ASCII-Nr. > 90 entstehen, muss der Verschiebeterm `(- 26 (- nr 65))` modulo 26 verwendet werden, woraus sich insgesamt

```
(define de-sw
  (lambda (sw)
    (integer-list->string
     (map
      (lambda (nr)
        (+ (modulo (- 26 (- nr 65)) 26) 65))
      (string->integer-list sw)))))
```

ergibt.
Bevorzugt man eine Entschlüsselungsfunktion mit eigenem Namen (s.o.), kann natürlich

```
(define de-vigenere
  (lambda (klartext sw)
    (vigenere klartext (de-sw sw))))
```

definiert werden.

3.2 Vernam 1: Bitweises Vigenère-Verfahren

Die Grundidee ist die gleiche wie beim Vigenère-Verfahren, lediglich die Repräsentation der Einzelzeichen und somit das konkrete Verschiebungsverfahren ist anders: Während bei Vigenère die Einzelverschiebung einer Addition modulo 26 entspricht,

$$\begin{aligned} K + I \rightarrow [(75-65)+(73-65)]_{mod\ 26} = (10+8)_{mod\ 26} \rightarrow 18+65 \rightarrow S \\ R + N \rightarrow [(82-65)+(78-65)]_{mod\ 26} = (17+13)_{mod\ 26} \rightarrow 4+65 \rightarrow E \\ Y + F \rightarrow [(89-65)+(70-65)]_{mod\ 26} = (24+5)_{mod\ 26} \rightarrow 3+65 \rightarrow D \\ usw. \end{aligned}$$

und bei der Entschlüsselung die Gegenschiebung einer Subtraktion modulo 26,

$$\begin{aligned} S - I \rightarrow [(83-65)-(73-65)]_{mod\ 26} = (18-8)_{mod\ 26} \rightarrow 10+65 \rightarrow K \\ E - N \rightarrow [(69-65)-(78-65)]_{mod\ 26} = (4-13)_{mod\ 26} \rightarrow 17+65 \rightarrow R \\ D - F \rightarrow [(68-65)-(70-65)]_{mod\ 26} = (3-5)_{mod\ 26} \rightarrow 24+65 \rightarrow Y \\ usw. \end{aligned}$$

werden jetzt die Klartextzeichen und die Schlüsselzeichen als Bitfolgen dargestellt, dann erfolgt für die Bits eine paarweise Addition modulo 2:

$$\begin{aligned} K + I \rightarrow (10 + 8)_{mod\ 26} &= (01010 + 01000)_{mod\ 2} = XOR(01010, 01000) = 00010 \\ R + N \rightarrow (17 + 13)_{mod\ 26} &= (10001 + 01001)_{mod\ 2} = XOR(10001, 01001) = 11000 \\ Y + F \rightarrow (24 + 5)_{mod\ 26} &= (11000 + 00101)_{mod\ 2} = XOR(11000, 00101) = 11101 \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad usw. \end{aligned}$$

Insgesamt wird also

```
KRY... -->  00010 11000 11101 ...
```

Auf bit-Ebene entspricht die Addition modulo 2 dem bekannten logischen „Entweder - oder" bzw. dem ausschließenden „ODER", auch XOR abgekürzt, wenn man die logischen Werte `#f` (falsch) und `#t` (wahr) durch 0 bzw. 1 ersetzt, und zwar mit den Eigenschaften

$$\begin{aligned} XOR(0, 0) = XOR(1, 1) = 0 \\ XOR(0, 1) = XOR(1, 0) = 1 \end{aligned}$$

Daraus ergibt sich der Vorteil, dass

$$XOR(XOR(a, b), b) = a$$

was bedeutet, das auf bit-Ebene mit dem gleichen Bit dechriffiert werden kann, mit dem auch chiffriert wurde!
Als maschinennahe Operation ist XOR in vielen Programmiersprachen implementiert, und zwar nicht nur logischen Operation oder Funktion, sonder auch bitweise oder auch byteweise. Da XOR in den Lernsprachen-Versionen von *DrScheme* einschließlich DMdA nicht vorgesehen ist, muss man es selbst definieren. Im Unterschied zum logischen XOR soll es `XOR-bit` heißen:

```
; XOR-bit: bit bit --> bit
(: XOR-bit (natural natural -> natural))
```

mit z.B.

```
(check-expect
 (XOR-bit 1 1)
 0)
(check-expect
 (XOR-bit 0 1)
 1)
```

also

```
(define XOR-bit
  (lambda (a b)
    (modulo (+ a b) 2)))
```

Bei der o.a. Codierung des Klartextes und des Schlüssels in Bitfolgen soll aus jedem Buchstaben eine Bitfolge der Länge 5 werden, und zwar aus folgenden Gründen:

1. In der Kryptologie ist es aus Gründen des Lesbarkeit üblich, Bitfolgen in 5-er Blöcke zu unterteilen
2. Es gibt 26 Buchstaben, deren Transformationen in die Nummern 1, 2, ..., 26 als Dualzahlen interpretierbare Bitfolgen von höchstens der Länge 5 ergeben ($26 = 11010$).

Das gesamte Verfahren im Überblick an einem einfachen Beispiel:

Chriffrierung				
Klartext (mit ASCII-Nr.)	A (65)	D (68)	A (65)	M (77)
KT als Dez.-Werte	1	4	1	13
KT als Bit-Blöcke	00001	00100	00001	01101
Schlüsselwort	E	V	A	
Schlüssel	E	V	A	E
Schlüssel als Bit-Blöcke	00101	10101	00001	00101
Chiffrat als Geheim„Text"	00100	10001	00000	01001
Dechiffrierung				
Schlüssel als Bit-Blöcke	00101	10101	00001	00101
„DeChiffrat" = KT als Bit-Blöcke	00001	00100	00001	01101
KT als Dez.-Werte	1	4	1	13
Klartext (mit ASCII-Nr.)	A (65)	D (68)	A (65)	M (77)

Es fällt auf, dass im Gegensatz zu allen vorangegangenen Verfahren der verschlüsselte Klartext als Geheimtext nicht aus einer Zeichenkette, sondern aus einer Bit-Folge besteht.
Dass liegt zum einen daran begründet, dass eine Umwandlung in ASCII-Zeichen das Entstehen von Nicht-Buchstaben zur Folge hätte, z.B. ergibt in obigem Beispiel eine bitweise Verschlüsselung von A mit A: $XOR(00001, 00001) = 000000$, was in der Rücktransformation durch Addition von 64 die ASCII-Nr. 64, also das Zeichen @ ergäbe; zum anderen genügen für eine elektronische Verarbeitung lediglich Bitfolgen.

Fassen wir zusammen: Das bit-weise Vigenère-Verfahren unterscheidet sich von dem klassischen Vigenère-Verfahren dadurch, dass bitweise verschlüsselt wird und das Ergebnis kein Geheimtext im Sinne einer Buchstabenfolge, sondern eine Reihe von 5-bit-Blöcken ist.
Berücksichtigt man diesen Umstand, bietet es sich an, einen solchen 5-bit-Block als Liste von Einzelbits ist – also 0 und 1 – zu modellieren und das gesamt Chiffrat wiederum als eine Liste von diesen Listen.
Daher werden folgende *Definitionen* vereinbart:

bit-liste liegt vor bei `(list` b_1 b_2 ... b_n`)` mit $b_i \in \{0, 1\}$

5bitliste liegt vor bei `(list` b_1 b_2 ... b_5`)` mit $b_i \in \{0, 1\}$

bitlisten-liste liegt vor bei `(list (list` b_{11} b_{12} ... b_{1n}`) (list` b_{21} b_{22} ... b_{2n}`) ...` `(list` b_{m1} b_{m2} ... b_{mn}`))` mit $b_{ij} \in \{0, 1\}$

5bitlisten-liste liegt vor bei `(list (list` b_{11} b_{12} ... b_{15}`) (list` b_{21} b_{22} ... b_{25}`) ...` `(list` b_{m1} b_{m2} ... b_{m5}`))` mit $b_{i5} \in \{0, 1\}$

zeichen-liste liegt vor bei `(list` z_1 z_2 ... z_n`)` mit $z_i \in$ {"A", "B", "C",..., "Z"}

Das ergibt folgenden Vertrag für unsere gesuchte Verschlüsselungsfunktion:

```
; vernam1: KT schluesselwort --> 5bitlisten-liste(als GT)
(: vernam1 (string string -> (list (list natural))))
```

mit z.B.

```
(check-expect (vernam1 "ADAM" "EVA")
        (list (list 0 0 1 0 0) (list 1 0 0 1 0) (list 0 0 0 0 0) (list 0 1 0 0 0)))
```

Das Funktionsgerüst

```
(define vernam1
  (lambda (klartext sw)
   ... <Vorbereitungen: klartext als 5bitlisten-liste, Länge dieser Liste> ...
      (vernam-hilfe ...))))
```

unterscheidet sich von dem von `vigenère` durch die Folgen der der bitweisen Verschlüsselung, d.h. im einzelnen

- die Klartext-Zeichenkette wird als 5bitlisten-Liste, etwa `"ADAM"` → `(list (list 0 0 0 0 1) (list 0 0 1 0 0) (list 0 0 0 0 1) (list 0 1 1 0 1))`, genannt `klartext-liste`, bereitgestellt, etwa durch eine Funktion `string->5bitlisten-liste`.

- die eigentliche Verschlüsselungsfunktion `vernam-hilfe` verbraucht diese Klartextliste zusammen mit dem Schlüsselwort (noch als Zeichenkette) und der Länge der Klartextliste, verschlüsselt – nachdem auch das Schlüsselwort in eine 5bitlisten-Liste umgewandelt wurde – mittels einer Funktion `XOR-bit-list` und gibt das Chiffrat als ebenfalls als 5bitlisten-Liste zurück

Entsprechend ergibt sich:

```
(define vernam1
  (lambda (klartext sw)
    (let
        ((klartext-liste (string->5bitlisten-liste (bereinige-klartext klartext))))
      (vernam-hilfe klartext-liste sw (length klartext-liste)))))
```

Wir entwickeln jetzt die fehlenden Funktionen:

Zunächst wird für

```
; string->5bitlisten-liste: KT --> 5bitlisten-liste
(: string->5bitlisten-liste (string -> (list (list natural))))
```

mit z.B.

```
(check-expect (string->5bitlisten-liste "ADAM")
   (list (list 0 0 0 0 1) (list 0 0 1 0 0) (list 0 0 0 0 1) (list 0 1 1 0 1)))
```

das Wesentliche in

```
(define string->5bitlisten-liste
  (lambda (zeichenkette)
    (zahlenliste->5bitlisten-liste (string->integer-list zeichenkette))))
```

durch

```
(: zahlenliste->5bitlisten-liste ((list integer) -> (list (list natural))))
```

mit z.B.

```
(check-expect
 (zahlenliste->5bitlisten-liste (list 65 68 65 78))
 (list (list 0 0 0 0 1) (list 0 0 1 0 0) (list 0 0 0 0 1) (list 0 1 1 1 0)))
```

erledigt, d.h. die Funktion hat die Aufgabe, die Liste der ASCII-Nr. der Zeichen in eine 5bitlisten-Liste umzuwandeln. Also muss die ASCII-Nr-Liste durchlaufen werden und für jedes Element ein eigene Liste mit 5 Bit zu erzeugen:

```
(define zahlenliste->5bitlisten-liste
  (lambda (liste)
    (cond
      ((empty? liste) empty)
      (else
       (cons ... <Erzeuge Liste mit 5 bit> ...
             (zahlenliste->5bitlisten-liste (rest liste)))))))
```

Diese Erzeugung übernimmt

```
; dez->dual5; ganzzahl --> 5bitliste
(: dez->dual5 (natural -> (list natural)))
```

mit z.B.

```
(check-expect
 (dez->dual5 13)
 (list 0 1 1 0 1))
(check-expect
 (dez->dual5 45)
```

```
 (list 0 1 1 0 1))
(check-expect
 (dez->dual5 77)
 (list 0 1 1 0 1))
```

Man erkennt an den drei Beispielen, dass die gesuchte Funktion eine Dezimalzahl in eine Liste von 5 bits umwandeln soll oder, anders gesagt, in eine Dualzahl der Länge 5, also werden nur die unteren 5 bits berücksicht, was Division modulo $32 = 2^5$ entspricht.
Damit ist auch eine Division modulo 64 erledigt, und man braucht von den ASCII-Nr. der Buchstaben (65, 66, ..., 90) nicht 64 zu subtrahieren, um die eingangs erwähnte Nummerierung zu erhalten. Zu diesem Zweck benötigt man in

```
(define dez->dual5
  (lambda (dezZahl)
    (reverse (dez->dual-hilfe dezZahl 5))))
```

(die nur die Reihenfolge umdreht) eine Hilfsfunktion

```
; dez->dual-hilfe: ASCII bitzahl --> bitliste
(: dez->dual-hilfe (natural natural -> (list natural)))
```

bei der der zweite Parameter `stellenzahl` die Anzahl der unteren Bits für die Umwandlung in eine Dualzahl in Form einer Liste der Bits in umgekehrter Reihenfolge (!) angibt, also etwa

```
(check-expect
 (dez->dual-hilfe 78 5)
 (list 0 1 1 1 0))
(check-expect
 (dez->dual-hilfe 78 6)
 (list 0 1 1 1 0 0))
(check-expect
 (dez->dual-hilfe 78 7)
 (list 0 1 1 1 0 0 1))
```

Zu diesem Zweck wird die gegebene Zahl mit Rest durch 2 geteilt und der Rest in eine Liste geschrieben, solange, bis durch den Parameter `stellenzahl` mit dem Wert Null eine Terminierung erreicht ist:

```
(define dez->dual-hilfe
  (lambda (zahl stellenzahl)
    (cond ((zero? stellenzahl) empty)
          (else
           (cons (remainder zahl 2)
                 (dez->dual-hilfe (quotient zahl 2) (- stellenzahl 1)))))))
```

Es bleibt noch die Definition der Hilfsfunktion `vernam-hilfe`, die – wie oben schon erwähnt – die 5bitlisten-Liste des Klartextes mit Hilfe des Schlüsselwortes (noch als Zeichenkette) sowie der Länge der Klartextliste verschlüsselt – nachdem auch das Schlüsselwort in eine 5bitlisten-Liste umgewandelt wurde – durch eine Funktion `XOR-bit-list` und das Chiffrat als ebenfalls als 5bitlisten-Liste zurückgibt:

```
; vernam-hilfe: 5bitlisten-liste(als KT) schluesselwort KT-Länge
;       --> 5bitlisten-liste(als GT)
(: vernam-hilfe ((list (list natural)) string natural -> (list (list natural))))
```

Um also wie im Beispiel "ADAM" mit "EVA" zu verschlüsseln, müssen also – nachdem das Schlüsselwort zum Schlüssel "EVAE" ergänzt wurde - die Zeichenketten in 5-bit-Listen-Form vorliegen:

```
"ADAM"
--> (list (list 0 0 0 0 1) (list 0 0 1 0 0) (list 0 0 0 0 1) (list 0 1 1 0 1))
"EVAE"
-->  (list (list 0 0 1 0 1) (list 1 0 1 1 0) (list 0 0 0 0 1) (list 0 0 1 0 1))
```

Wenn man die jeweiligen Bit-Paare mit XOR verknüpft, ergibt sich

```
(list (list 0 0 1 0 0) (list 1 0 0 1 0) (list 0 0 0 0 0) (list 0 1 0 0 0))
```

Also erwartet man

```
(check-expect
 (vernam-hilfe
  (list (list 0 0 0 0 1) (list 0 0 1 0 0) (list 0 0 0 0 1) (list 0 1 1 0 1))
  "EVA"
  4)
 (list (list 0 0 1 0 0) (list 1 0 0 1 0) (list 0 0 0 0 0) (list 0 1 0 0 0)))
```

Dabei stammt der Wert 4 als Länge des KT zum dritten Parameter von `vernam-hilfe`,der von `vernam1` übernommen und im Folgenden benötigt wird. Da das Schlüsselwort auch als 5bitlisten-Liste vorliegen muss mit samt deren Länge, ergibt sich als Gerüst:

```
(define vernam-hilfe
  (lambda (liste sw laenge-KT)
    (let ((sw-liste (string->5bitlisten-liste sw)))
   ...<eigentliche Verschlüsselung mittels XOR>...)))
```

Dabei wird die 5bitlisten-Liste des Klartextes durchlaufen und eine einzelne 5-bit-Liste mit einer einzelnen 5-bit-Liste des Schlüssels mit Hilfe von `XOR-bit-list` verknüpft als Element einer der Ergebnisliste ausgegeben:

```
(define vernam-hilfe
  (lambda (liste sw laenge-KT)
    (let ((sw-liste (string->5bitlisten-liste sw)))
      (cond
        ((empty? liste) empty)
        (else
         (cons
          (XOR-bit-list
           ...<5-bit der KT-Liste>...<5-bit der SW-Liste>...))
          (vernam-hilfe (rest liste) sw laenge-KT)))))))
```

Um zwei bit-Listen gleicher Länge mit XOR zu verknüpfen, benötigt man eine Funktion

```
; XOR-bit-list: bitliste bitliste --> bitliste
(: XOR-bit-list ((list natural) (list natural) -> (list natural)))
```

mit etwa

```
(check-expect
 (XOR-bit-list (list 1 0 1) (list 1 0 0))
 (list 0 0 1))
```

Dazu greifen wir auf die bereits definierte Funktion `XOR-bit` zurück und wenden sie mittels `map` auf zwei (gleichlange) Listen an:

```
(define XOR-bit-list
  (lambda (liste1 liste2)
    (map XOR-bit liste1 liste2)))
```

Den ersten Operanden liefert aufgrund des Listendurchlaufs jeweils `(first liste)` , der zweite Operand muss die zugehörige string->5bitlisten-liste des Schlüssels sein: Im Beispiel wird zuerst das `"A"` von „ADAM" mit dem `"E"` von „EVA" verknüpft, also `(XOR-bit-list (list 0 0 0 0 1) (list 0 0 1 0 1))` ausgewertet. Da das Schlüsselwort kürzer als der Klartext ist, muss z.B. für das `"M"` von „ADAM" wieder das erste Zeichen des Schlüsselwortes, als das`"E"` als Verknüpfungspartner bereitgestellt werden, also im Wechsel das 1., das 2. und 3. Zeichen des Schlüssels (als 5-bit-Liste). Dies geschieht, indem durch `list-ref` aus der string->5bitlisten-liste `sw-liste` mittels `(modulo (- laenge-KT (length liste)) laenge-sw)` das 0., das 1. und das 2. Element ermittelt.
Insgesamt ergibt sich so:

```
(define vernam-hilfe
  (lambda (liste sw laenge-KT)
    (let ((sw-liste (string->5bitlisten-liste sw)))
      (cond
        ((empty? liste) empty)
        (else
         (cons
```

```
          (XOR-bit-list
           (first liste)
           (list-ref
            sw-liste (modulo (- laenge-KT (length liste)) (lenght sw-liste)))
          (vernam-hilfe (rest liste) sw laenge-KT)))))))
```

Im Unterschied zu den vorangegangenen Verfahren liefert `vernam1` den Geheimtext nicht als Zeichenkette, sondern als Liste von 5-bit-Listen. Dies ist für den legalen Empfänger ausreichend, da hier von maschineller Chiffrierung und Dechiffrierung ausgegangen wird.
Um diese bitweise vorliegende Nachricht verstehen zu können, benötigt er natürlich eine Umwandlungsfunktion

```
(: 5bitlisten-liste->string ((list (list natural)) -> string))
```

die eine lesbare Zeichenkette erzeugt, z.B. mit

```
(check-expect
 (5bitlisten-liste->string
  (list (list 0 0 0 0 1) (list 0 0 1 0 0) (list 0 0 0 0 1) (list 0 1 1 0 1)))
 "ADAM")
```

und dem Gerüst

```
(define 5bitlisten-liste->string
  (lambda (bit5liste)
    (integer-list->string
     ......... bit5liste......))))
```

Zunächst muss jeder 5-bit-Block in eine Dezimalzahl umgewandelt werden, was mit

```
; dual->dez-hilfe: 5bitliste --> ganzzahl
(: dual->dez-hilfe ((list natural) -> natural))
```

geschehen kann, etwa

```
(check-expect
 (dual->dez-hilfe (list 0 0 1 1 0))
 6)
```

Dazu muss die Bit-Liste durchlaufen werden, zu jedem Bit die Zweierpotenz mittels `(expt 2 (- (length liste) 1))` gebildet und mit dem jeweiligen Bit `(first liste)` multipliziert werden; die Summe liefert das Ergebnis:

```
(define dual->dez-hilfe
  (lambda (liste)
    (cond ((empty? liste) 0)
          (else (+ (* (first liste) (expt 2 (- (length liste) 1)))
                   (dual->dez-hilfe (rest liste)))))))
```

Damit man die ASCII-Nummern der Zeichen erhält, muss zu den so erhaltenen Dezimalzahlen noch 64 addiert werden, was einfach mittels

```
(define add64
  (lambda (zahl)
    (+ zahl 64)))
```

erreicht wird.
Beide Funktionen, also `dual->dez-hilfe` und `add64` werden nacheinander mittels `map` auf die vorliegende `bit5liste` angewendet und man erhält insgesamt

```
(define 5bitlisten-liste->string
  (lambda (bit5liste)
    (integer-list->string
     (map add64
          (map dual->dez-hilfe bit5liste)))))
```

An dem Aufruf

```
(5bitlisten-liste->string (vernam1 "ADAM" "EVA"))   -->  "DR@H"
```

sieht man ohnehin, dass durch die XOR-Operationen auch Zeichen außerhalb des Alphabets entstehen, d.h. es besteht keine Notwendigkeit, den Geheimtext als Zeichenkette lesbar zu machen, wie bereits oben erwähnt, vielmehr muss der **entschlüsselte** Geheimtext als Zeichenkette lesbar sein, wie wir im Folgenden sehen:

Entschlüsselung

Aufgrund der Eigenschaft $XOR(XOR(a,b),b) = a$ ist eine Entschlüsselungsroutine

```
; de-vernam1: 5bitlisten-liste(als GT) schluesselwort --> KT
(: de-vernam1 ((list (list natural)) string -> string))
```

mit z.B.

```
(check-expect
 (de-vernam1
  (list (list 0 0 1 0 0) (list 1 0 0 1 0) (list 0 0 0 0 0) (list 0 1 0 0 0)) "EVA")
   "ADAM")
(check-expect
 (de-vernam1 (vernam1 "ADAM" "EVA") "EVA")
 "ADAM")
```

schnell gefunden: auf die Liste der 5bitlisten-Liste des Geheimtextes, etwa `geheimtext-liste` wird wieder `vernam-hilfe` (was das bitweise XOR bewirkt) zusammen mit dem Schlüssel angewendet und anschließend mittels `5bitlisten-liste->string` in lesbare Zeichenketten umgewandelt:

```
(define de-vernam1
  (lambda (geheimtext-liste sw)
    (letrec
        ((laenge-KT-liste (length geheimtext-liste)))
      (5bitlisten-liste->string (vernam-hilfe geheimtext-liste sw laenge-KT-liste)))))
```

3.3 Vernam 2: Einmal-Schablone (One-Time-Pad)

Ein besonders sichere Variante der XOR-Verschlüsselung auf Bit-Ebene stellt das Verfahren mit *Einmal-Schablone* oder *One-Time-Pad* dar: der Schlüssel ist eine Zufalls-Bitfolge mit der gleichen Länge wie die Klartext-Bitliste. Dabei darf der Schlüssel aus Sicherheitsgründen nur einmal benutzt werden (auf die Problematik computer-erzeugter Pseudo-Zufallszahlen anstatt echter Zufallszahlen wird hier nicht eingegegangen).
Angenommen, es soll

```
"ADAM" --> (list (list 0 0 0 0 1) (list 0 0 1 0 0) (list 0 0 0 0 1) (list 0 1 1 0 1))
```

mit dem gleichlangen Zufallsschlüssel

```
ein-zs --> (list (list 0 0 0 1 1) (list 0 1 0 1 0) (list 1 1 0 0 0) (list 1 0 1 1 1))
```

verschlüsselt werden. Dann erwarten wir von der gesuchten Funktion

```
;vernam2: KT 5bitlisten-liste -> 5bitlisten-liste
(: vernam2 (string (list (list natural)) -> (list (list natural))))
```

z.B. folgendes:

```
(check-expect
 (vernam2 "ADAM" ein-zs)
 (list (list 0 0 0 1 0) (list 0 1 1 1 0) (list 1 1 0 0 1) (list 1 1 0 1 0)))
(check-expect
 (vernam2 (5bitlisten-liste->string (vernam2 "ADAM" ein-zs)) ein-zs)
 (list (list 0 0 0 0 1) (list 0 0 1 0 0) (list 0 0 0 0 1) (list 0 1 1 0 1)))
(check-expect
 (bit5liste->text (vernam2 (5bitlisten-liste->string (vernam2 "ADAM" ein-zs)) ein-zs))
 "ADAM")
```

Das Gerüst ist ähnlich wie bei `vernam1` mit dem Unterschied, dass der zweite Parameter `zs` keine Zeichenkette ist, sondern die 5bitlisten-Liste der Zufallsbits:

```
(define vernam2
  (lambda (klartext zs)
 ...))
```

Zunächst wird der Klartext als 5bitlisten-Liste als lokale Variable bereitgestellt.
Will man sicher gehen, dass bei verschieden langen Klartext-Listen und Schlüssel-Listen kein Laufzeitfehler entsteht, kann man eine Fallunterscheidung einbauen, die dafür sorgt, dass in diesem Fall die Verschlüsselung nicht versucht wird und stattdessen eine leere Liste ausgegeben wird. Im positiven Fall führt die Hilfsfunktion `vernam2-hilfe` die eigentliche Hilfsfunktion durch:

```
(define vernam2
  (lambda (klartext zs)
    (let
        ((klartext-liste (string->5bitlisten-liste (bereinige-klartext klartext))))
      (cond
        ((equal? (length klartext-liste) (length zs))
         (vernam2-hilfe klartext-liste zs))
        (else
         empty)))))
```

Die Hilfsfunktion

```
; vernam2-hilfe: bitlisten-list bitlisten-liste --> bitlisten-liste
(: vernam2-hilfe
   ((list (list natural)) (list (list natural)) -> (list (list natural))))
```

durchläuft zwei gleichlange bitlisten-Listen und verknüpft die Blöcke jeweils mit `XOR-bit-list` , z.B.

```
(check-expect
 (vernam2-hilfe (list (list 1 0)) (list (list 1 1)))
 (list (list 0 1)))
```

Das Ergebnis ist wieder eine bitlisten-Liste, die die Elemente der Ergebnisliste bilden, die dadurch entsteht, dass sich die Funktion mit den Resten der beiden Listen selbst aufruft:

```
(define vernam2-hilfe
  (lambda (liste1 liste2)
    (cond
      ((empty? liste1) empty)
      (else
       (cons (XOR-bit-list (first liste1) (first liste2))
             (vernam2-hilfe (rest liste1) (rest liste2)))))))
```

Entschlüsselung

Da auch `vernam2` wie `vernam1` den Geheimtext als 5bitlisten-Liste liefert, muss zur Entschlüsselung wieder mit `5bitlisten-liste->string` das Chiffrat in eine Zeichenkette umgewandelt werden, aus der `vernam2` wieder den ursprünglichen Klartext als Liste liefert wird, wie man an

```
(check-expect
 (vernam2 (5bitlisten-liste->string (vernam2 "ADAM" ein-zs)) ein-zs)
 (list (list 0 0 0 0 1) (list 0 0 1 0 0) (list 0 0 0 0 1) (list 0 1 1 0 1)))
```

sieht.

Übungen

1. Bit-Blöcke werden zur visuellen Darstellung i.a. nicht als Liste, sondern z.B. als Zeichenketten repräsentiert, also z.B. `001101` statt `(list 0 0 1 1 0 1)`
   a) Entwickle eine Funktion `bitliste->bitstring`, die eine Bit-Liste beliebiger Länge in eine Zeichenkette umwandelt

b) Entwickle eine Funktion `5bitlisten-liste->bitstring`, die mit Hilfe von `bitliste->bitstring` eine Liste von Bit-Listen in eine Zeichenkette umwandelt. Versuche, dabei `apply` und `map` einzusetzen
c) Entwickle zu a) und b) Umkehrfunktionen

4 Transposition

4.1 Tabellentransposition, allgemein

Im Gegensatz zu den Substitutionsverfahren werden Buchstaben nicht durch andere Buchstaben ersetzt, stattdessen ändert sich nur ihre Position innerhalb des Wortes. Wenn die so erzeugt neue Zeichenkette als Wort oder Satz einen Sinn hat, wird es auch als *Anagramm* bezeichnet.

Beispiele:

```
Eisbaer --> Abreise
amerikanisch --> Irak Maschine
Dieter Bohlen --> Bloedenhirte
```

Mathematisch gesehen handelt es sich um Permutationen einer Zeichenkette der Länge n, entsprechend gibt es $n!$ Möglichkeiten.
Allerdings wird die mit n extrem wachsende Anzahl der Möglichkeiten ($20! = 2432902008176640000$) dadurch stark eingeschränkt, das das Transpositionsverfahren nach einem einfachen Algorithmus erfolgen muss, damit der Empfänger den Geheimtext entschlüsseln kann.

Im Folgenden werden einige Verfahren codiert, die zu den sog. *Tabellen-Transpostionen* gehören. Dabei wird der Klartext in die Felder eine Tabelle der Breite k geschrieben.

Als Beispiel soll der Klartext „Informatik macht Freude" dienen. Mit der Breite $k = 4$ ergibt sich:

```
KT: INFORMATIKMACHTFREUDE  -->
```

I	N	F	O
R	M	A	T
I	K	M	A
C	H	T	F
R	E	U	D
E			

An dieser Stelle stellt sich die Frage nach der Datenmodellierung: Da der Datentyp `vector` in den DMdA-Sprachen nicht zu Verfügung steht, bietet es sich an, die Tabelle als Liste von Zeilen zu interpretieren, wobei jede Zeile selbst wieder eine Liste ist, die ihrerseits aus den Buchstaben des Klartextes in Form von Ein-Zeichen-Zeichenketten ist:

```
(list
 (list "I" "N" "F" "O")
 (list "R" "M" "A" "T")
 (list "I" "K" "M" "A")
 (list "C" "H" "T" "F")
 (list "R" "E" "U" "D")
 (list "E"))
```

Der zweite Schritt ist das Auslesen der Tabelle. Die verschiedenen Auslese-Verfahren bilden zusammen mit der Spaltenzahl k den *Schlüssel* der jeweiligen Tabellen-Transposition.

Allen Tabellen-Transpositionen ist das Einlesen des Klartextes gemeinsam, daher wird zuerst eine Funktion

```
; kt->tabelle: KT spaltenbreite --> tabelle
(: kt->tabelle (string natural -> (list (list string))))
```

entwickelt. Dabei gibt der zweite Parameter die Spaltenzahl an:

```
(check-expect (kt->tabelle "INFORMATIKMACHTFREUDE" 4)
              (list
               (list "I" "N" "F" "O")
               (list "R" "M" "A" "T")
               (list "I" "K" "M" "A")
               (list "C" "H" "T" "F")
```

```
               (list "R" "E" "U" "D")
               (list "E")))
```

Zunächst wird der Klartext mit `bereinige-klartext` vorbereitet und mit `string->strings-list` in eine Liste von Einzelzeichen umgewandelt, die als lokale Variable namens `kt-liste` bereitgestellt wird:

```
(string->strings-list "INFORMATIKMACHTFREUDE") -->
  (list "I" "N" "F" "O" "R" "M" "A" "T" "I" "K"
  "M" "A" "C" "H" "T" "F" "R" "E" "U" "D" "E")
```

Man erhält als Schablone:

```
(define kt->tabelle
  (lambda (klartext k)
      ...<Alle Zeichen von
      (string->strings-list (bereinige-klartext klartext))
      einlesen>...))
```

Zunächst benötigt man eine Funktion

```
; zeile-einlesen: KT-liste spaltenbreite zähler --> zeile(als zeichenliste)
(: zeile-einlesen ((list string) natural natural -> (list string)))
```

die die ersten k Zeichen einer Zeichenliste als Liste ausgibt und dabei eine Liste der Zeichen `kt` , die Anzahl der Spalten k und einen Zähler `x` verbraucht, der von 0 bis k hochzählt, also z.B.

```
(check-expect
 (zeile-einlesen (list "A" "D" "A" "M") 3 0)
 (list "A" "D" "A"))
(check-expect
 (zeile-einlesen (list "I" "N" "F" "O" "R" "M" "A" "T" "I" "K") 4 0)
 (list "I" "N" "F" "O"))
```

Die Liste `kt-liste` wird rekursiv durchlaufen, und der Zähler von 0 aus bei jedem Selbstaufruf um 1 erhöht, also ergibt sich die Schablone

```
(define zeile-einlesen
  (lambda (kt-liste k x)
    (cond
      (...<Terminierung>... empty)
      (else
       (cons (first kt-liste) (zeile-einlesen (rest kt-liste) k (+ x 1)))))))
```

Offensichtlich ist die Terminierung dann erreicht, wenn keine weiteren Elemente mehr hinzuzufügen sind, also die gewünschte Anzahl der Elemente erreicht ist d.h. x=k, oder wenn keine Elemente mehr in `kt-liste` vorhanden sind, also wenn `kt-liste` leer ist.
Insgesamt ergibt sich daraus:

```
(define zeile-einlesen
  (lambda (kt-liste k x)
    (cond
      ((or (equal? k x) (empty? kt-liste)) empty)
      (else
       (cons (first kt-liste) (zeile-einlesen (rest kt-liste) k (+ x 1)))))))
```

Um nun die ganze Liste `kt-liste` in Einzellisten der Länge k oder weniger, wenn ein Rest übrig bleibt zu zerlegen, wird eine weitere Funktion

```
; alle-einlesen: KT-liste spaltenbreit zähler --> tabelle
(: alle-einlesen ((list string) natural natural -> (list (list string))))
```

benötigt, die mit Hilfe von `zeile-einlesen` eine Liste von Listen, die die Zeilen der Tabelle repräsentieren, erzeugt; da die Liste `kt-liste` rekursiv durchlaufen wird, sind dabei sind folgende Fälle zu unterscheiden:

1. wenn `kt-liste` leer ist, finden keine Selbstaufrufe statt, also ist die Erzeugung der Ausgabeliste beendet, d.h. es wird `empty` geliefert.

2. wenn der Zähler `y` ein Vielfaches von `k` ist, d.h. also $y = 0, k, 2k, 3k, ...$ wird mittels `zeile-einlesen` eine Einzelliste mit `k` Elementen erzeugt und als neues Element der Ergebnisliste hinzugefügt und die Funktion ruft sich mit dem Rest von `kt-liste` und erhöhtem Zähler selbst auf
3. wenn der Zähler `y` kein Vielfaches von `k` ist, ruft sich `alle-einlesen` einfach nur selbst auf mit dem nächsten Element von `kt-liste`

Zunächst ergibt sich daraus, dass die Funktion neben `kt-liste` und der Spaltenzahl als dritten Parameter einen Zähler, etwa `y` verbraucht, der bei jedem Selbstaufruf erhöht wird:

```
(define alle-einlesen
  (lambda (kt-liste k y)
    (cond
      ((empty? kt-liste) empty)
      ((zero? (modulo y k)) ...(+ y 1)...)
      (else (alle-einlesen (rest kt-liste) k (+ y 1)
        ))))
```

Die verbleibende Ellipse in der Schablone ist schnell gefüllt: Das neue Element der Ausgabeliste wird durch den Aufruf `(zeile-einlesen kt-liste k 0)` erzeugt (wobei der dortige Zähler x mit 0 beginnt) und mit `(cons ...)` der Ausgabeliste hinzugefügt. Der Rest ergibt sich durch den Selbstaufruf `(alle-einlesen (rest kt-liste) k (+ y 1))))` mit dem Rest von `kt-liste` und erhöhtem Zähler:

```
(define alle-einlesen
  (lambda (kt-liste k y)
    (cond
      ((empty? kt-liste) empty)
      ((zero? (modulo y k))
       (cons (zeile-einlesen kt-liste k 0) (alle-einlesen (rest kt-liste) k (+ y 1))))
      (else (alle-einlesen (rest kt-liste) k (+ y 1) )))))
```

Wenn die Länge des (bereinigten) Klartextes nicht gerade ein Vielfaches der von k ist, bleibt die unterste Zeile der Tabelle unvollständig in dem Sinne, dass die Länge der letzten Liste $< k$ ist. Wie man unten sehen wird, ist es für das Auslesen der Tabelle günstiger, wenn sie vollständig ist in dem Sinne, dass alle Zeilen (Listen) die gleiche Länge haben; in diesem Fall kann man auch von einer *Matrix* sprechen. Deshalb ist eine Funktion

```
; auffuellen: tabelle spaltenbreite --> tabelle
(: auffuellen ((list (list string)) natural -> (list (list string))))
```

zu konstruieren, die die letzte Liste ggfs. mit leeren Strings auffüllt:

```
(list                                    (list
 (list "I" "N" "F" "O")                   (list "I" "N" "F" "O")
 (list "R" "M" "A" "T")                   (list "R" "M" "A" "T")
 (list "I" "K" "M" "A")                   (list "I" "K" "M" "A")
 (list "C" "H" "T" "F")        —>         (list "C" "H" "T" "F")
 (list "R" "E" "U" "D")                   (list "R" "E" "U" "D")
 (list "E"))                              (list "E" "" "" ""))
```

Die gesuchte Funktion verbraucht die (unvollständige) Tabelle und die Spaltenbreite k:

```
(define auffuellen
  (lambda (tabelle k)
        .......))
```

Es muss zwischen zwei Fällen unterschieden werden, nämlich zwischen vollständiger und unvollständiger Tabelle:

```
(define auffuellen
  (lambda (tabelle k)
    (cond
      (<Matrix bzw. Tabelle unvollständig?> .............. ))))
```

```
      (else
       ..............))))
```

Im ersten Fall ist die letzte Liste kürzer als k, d.h. `(< (length (first (reverse tabelle))) k)` . Dann muss die letzte Zeile durch eine vervollständigte Zeile ersetzt werden, also wird an die Tabelle ohne letzte Liste, die man durch `(reverse (rest (reverse tabelle)))` erhält, eine neue vollständige Liste angehängt:

```
(append (reverse (rest (reverse tabelle))) (list <neue vollständige Liste>))
```

Ist die Matrix bereits vollständig, wird sie unverändert zurückgegeben:

```
(define auffuellen
  (lambda (tabelle k)
    (cond
      ((< (length (first (reverse tabelle))) k)
       (append (reverse (rest (reverse tabelle)))
               (list (zeile-auffuellen .......)))
      (else
       matrix))))
```

Die zum Auffüllen einer einzelnen Liste gesuchte Funktion

```
; zeile-auffuellen: zeichen-liste anzahl --> zeichenliste
(: zeile-auffuellen ((list string) natural -> (list string)))
```

verbraucht eine einfache Liste von Einzelzeichen und die Anzahl der benötigten leeren Strings:

```
(define zeile-auffuellen
  (lambda (eine-liste anzahl)
        ........
```

Z.B.

```
(check-expect (zeile-auffuellen (list "A") 3) (list "A" "" "" ""))
```

Die Funktion ruft sich selbst auf mit `anzahl` als Zähler, der so lange vermindert wird, bis er den Wert Null hat, was als Terminierung bedeutet, dass die Liste nicht mehr verändert wird:

```
(define zeile-auffuellen
  (lambda (eine-liste anzahl)
    (cond
      ((zero? anzahl) eine-liste)
      (else
       .......... (zeile-auffuellen eine-liste (- anzahl 1)) ........ )))))
```

Bei jedem Selbstaufruf wird mit `append` die neue Zeile an die bisherige Liste `(list "")` angehängt:

```
(define zeile-auffuellen
  (lambda (eine-liste anzahl)
    (cond
      ((zero? anzahl) eine-liste)
      (else
       (append (zeile-auffuellen eine-liste (- anzahl 1)) (list ""))))))
```

Es bleibt zu klären, mit welchen Werten `zeile-auffuellen` in `auffuellen` aufgerufen wird. Beim ersten Parameter, also der Liste, handelt es sich um die letzte Liste der Tabelle, also `(first (reverse tabelle))`, beim zweiten Parameter um die Anzahl der leeren Strings, also der Differenz zwischen k und der aktuellen Länge der letzten Liste: `(- k (length (first (reverse tabelle))))`.
Insgesamt ergibt sich:

```
(define auffuellen
  (lambda (tabelle k)
    (cond
      ((< (length (first (reverse tabelle))) k)
       (append (reverse (rest (reverse tabelle)))
               (list (zeile-auffuellen
```

```
                    (first (reverse tabelle))
                    (- k (length (first (reverse tabelle))))))))
      (else
       tabelle))))
```

Damit ist auch `kt->tabelle` komplett:

```
(define kt->tabelle
  (lambda (klartext k)
    (auffuellen
     (alle-einlesen
      (string->strings-list (bereinige-klartext klartext)) k 0) k)))
```

Unter den zahlreichen Möglichkeiten, eine Zeichentabelle auszulesen, werden hier nur ein paar einfache Varianten vorgestellt.

4.2 Einfache Spaltentransposition („Würfel")

Die einfachste und zugleich bekannteste Variante ist der sog. „Würfel"; dabei werden die Spalten der Matrix von links nach rechts ausgelesen, und zwar von oben nach unten:

"INFORMATIKMACHTFREUDE" -->

I	N	F	O
R	M	A	T
I	K	M	A
C	H	T	F
R	E	U	D
E			

--> "IRICRENMKHEFAMTUOTAFD"

Es wird also eine Funktion

```
; wuerfeleinfach: KT spaltenbreite --> GT
(: wuerfeleinfach (string natural -> string))
```

gesucht, die den Klartext als Zeichenkette und die gewünschte Tabellenbreite k verbraucht und den Geheimtext ebenfalls als Zeichenkette liefert, z.B.

```
(check-expect
 (wuerfeleinfach "INFORMATIKMACHTFREUDE" 4)
 "IRICRENMKHEFAMTUOTAFD")
(check-expect
 (wuerfeleinfach "INFORMATIKMACHTFREUDE" 6)
 "IACUNTHDFITEOKFRMRMAE")
```

Aufgrund der Datenmodellierung der Tabelle/Matrix als Liste von Zeichenketten-Listen besteht die Verschlüsselung aus drei Schritten:

1. Umwandlung der Zeichenkette des Geheimtextes in eine Liste von Zeichenketten-Listen (Tabelle) mittels `kt->tabelle`, was einem zeilenweisen Einlesen entspricht.
2. Spaltenweises Auslesen der Tabelle durch Umwandlung der Tabelle in eine neue Tabelle, deren Breite der Anzahl der Zeilen der ursprünglichen Tabelle entspricht:

```
(list                                          (list
 (list "I" "N" "F" "O")                         (list "I" "R" "I" "C" "R" "E")
 (list "R" "M" "A" "T")                         (list "N" "M" "K" "H" "E" "")
 (list "I" "K" "M" "A")     Transponieren       (list "F" "A" "M" "T" "U" "")
 (list "C" "H" "T" "F")     ------------>       (list "O" "T" "A" "F" "D" ""))
 (list "R" "E" "U" "D")
 (list "E" "" "" ""))
```

Dies kann dadurch erreicht werden, dass die Tabelle, wenn sie vollständig ist, also eine Matrix, transponiert wird, d.h. an der Diagonalen von links oben nach rechts unten gespiegelt wird.

3. Zeilenweises Auslesen der transponierten Matrix/Tabelle und Umwandlung in eine Zeichenkette, den GT

Im ersten Schritt wird die oben definierte Funktion `kt->tabelle` mit der Tabellenbreite k auf den Klartext angewandt und das Ergebnis als lokale Variablen `matrix` definiert:
Als Gerüst haben wir dann:

```
(define wuerfeleinfach
  (lambda (klartext k)
   ..... ))
```

Mit den oben erwähnten drei Arbeitsschritten ergibt sich als Schablone:

```
(define wuerfeleinfach
  (lambda (klartext k)
    (<zeilenweise Auslesen>
                (transponiere <eingelesene Tabelle>))))
```

Im Mittelpunkt dieses Verfahrens steht die Funktion

```
; transponiere: (vollständige)Tabelle -->  (vollständige)Tabelle
(: transponiere ((list (list %a)) -> (list (list %b))))
```

mit dem Gerüst

```
(define transponiere
  (lambda (matrix)
    ..... ))
```

und z.B. Erwartungen wie

```
(check-expect
 (transponiere
  (list
   (list 1 2 3)
   (list "a" "b" "c")))
 (list
  (list 1 "a")
  (list 2 "b")
  (list 3 "c")))

(check-expect
 (transponiere
  (list
   (list "I" "N" "F" "O")
   (list "R" "M" "A" "T")
   (list "I" "K" "M" "A")
   (list "C" "H" "T" "F")
   (list "R" "E" "U" "D")
   (list "E" "" "" ""))
  (list
   (list "I" "R" "I" "C" "R" "E")
   (list "N" "M" "K" "H" "E" "")
   (list "F" "A" "M" "T" "U" "")
   (list "O" "T" "A" "F" "D" ""))))
```

Die gesuchte Funktion muss also eine mxn-Matrix, repräsentiert als Liste von Listen, in eine nxm-Matrix, ebenfalls als Listen von Listen repräsentiert, umwandeln. Dabei hilft folgende Überlegung: Die *Higher-Order*-Funktion `map` verbraucht eine n-stellige Funktion $n > 1$ und n gleichlange Listen und liefert eine Liste der Ergebnisse, die durch Anwendung der Funktion auf die Elemente an jeweils gleicher Position in den Listen, wie z.B.

```
(map odd? (list -2 3 -4))
(map + (list 1 2 3) (list 4 5 6) (list 4 5 6))
```

Da `list` selbst auch eine Funktion ist, ist auch

```
(map list (list 1 2 3) (list "a" "b" "c"))
         --> (list (list 1 "a") (list 2 "b")  (list 3 "c"))
```

möglich, indem `list` zunächst die beiden ersten Elemente der Listen, dann die beiden zweiten Elemente usw. miteinander zu einer neuen Liste verknüpft.
Allerdings kann bei diesem Aufruf keine Matrix als Liste von Listen verbraucht werden, sondern lediglich zwei Einzellisten. Um dieses letzte Problem zu lösen, betrachten wir den Aufruf

```
(apply map (list list (list 1 2 3) (list "a" "b" "c")))
        --> list (list 1 "a") (list 2 "b")  (list 3 "c")))
```

Durch die Ersetzung und Verallgemeinerung

```
(list list (list 1 2 3) (list "a" "b" "c"))
        = (cons list (list (list 1 2 3) (list "a" "b" "c")))
        = (cons list <matrix>)
```

ergibt sich die komplette Funktion:

```
(define transponiere
  (lambda (matrix)
    (apply map (cons list matrix))))
```

Das Argument für den Aufruf von `transponiere` in o.a. Schablone für `wuerfeleinfach` erhalten wir durch `(kt->tabelle (bereinige-klartext klartext) k)`.
Der Funktionswert von

```
(transponiere (kt->tabelle (bereinige-klartext klartext) k))
```

liefert als transponierte Matrix wieder eine Liste von Zeichenlisten und kann somit folgendermaßen zeilenweise ausgelesen werden:
Mit `(map strings-list->string <Matrix>)` erhält man eine Liste von Zeichenketten, die durch `(apply string-append <Liste von Zeichenketten>)` zu einer Zeichenkette verknüpft werden.
Insgesamt ergibt sich:

```
(define wuerfeleinfach
  (lambda (klartext k)
    (apply string-append
           (map strings-list->string
                (transponiere (kt->tabelle (bereinige-klartext klartext) k))))))
```

Entschlüsselung

Es ist offensichtlich, dass `wuerfeleinfach` auch zur Dechiffrierung benutzt werden kann, wenn die Geheimtexttabelle ohne Auffüllung mit leeren Zeichen eine echte Matrix ergibt, d.h. wenn die Länge n des (bereinigten) Klartextes ein ganzzahliges Vielfaches der Spaltenbreite k ist, für die Dechiffrierung muss lediglich $k' = n/k$ als zweiter Parameter genommen werden.
In allen anderen Fällen funktioniert die Einlesefunktion `kt->tabelle` nicht mehr, da in der GT-Tabelle die leeren Zeichen in der letzten Spalte stehen, während sie in der KT-Tabelle in der letzten Zeile stehen.
Es muss also in der gesuchten Funktion

```
; de-wuerfeleinfach: GT spaltenbreite --> KT
(: de-wuerfeleinfach (string natural -> string))
```

mit z.B.

```
(check-expect
 (de-wuerfeleinfach "IRICRENMKHEFAMTUOTAFD" 4)
 "INFORMATIKMACHTFREUDE")
```

eine neue Einleseprozedur gefunden werden, ansonsten hat `de-wuerfeleinfach` die gleiche Struktur wie `wuerfeleinfach`:

```
(define de-wuerfeleinfach
  (lambda (geheimtext k)
    (apply string-append
           (map strings-list->string
                (transponiere
                 (<GT-Einlesen> (bereinige-klartext geheimtext)) k))))))
```

Wenn wir den oben beschriebenen Weg der Verschlüsselung unter Benutzung der Funktion `transponiere` rückwärts durchlaufen wollen, brauchen wir eine Einlesefunktion

```
; gt-liste-einlesen: zeichenliste spaltenbreite --> tabelle(als zeichenliste)
(: gt-liste-einlesen ((list string) natural -> (list (list string))))
```

die z.B.

```
(check-expect
 (gt-liste-einlesen (string->strings-list (bereinige-klartext "IRICRENMKHEFAMTUOTAFD")) 4)
 (list
        (list "I" "R" "I" "C" "R" "E")
        (list "N" "M" "K" "H" "E" "")
        (list "F" "A" "M" "T" "U" "")
        (list "O" "T" "A" "F" "D" "")))
```

bewirkt.
Bei dem zugehörigen Gerüst

```
(define gt-liste-einlesen
  (lambda (gt-liste k)
    ..... ))
```

ist zu beachten, dass der zweite Parameter k als Teil des Schlüssels die Spaltenbreite der KT-Tabelle, nicht aber die hier nötige Spaltenbreite der GT-Tabelle angibt.
Dabei sind - wie bereits oben erwähnt - zwei Fälle zu unterscheiden:

1. Ist die KT-Tabelle vollständig, d.h. eine Matrix ohne leere Zeichen, gilt für die Spaltenbreite k_{de} der GT-Tabelle: $k_{de} = n/k$, wobei n die Anzahl der Zeichen bzw. die Länge des GT angibt.
2. Ist die KT-Tabelle nicht vollständig, ist auch die GT-Tabelle nicht vollständig, d.h. die letzte Spalte der GT-Tabelle muss leere Zeichen enthalten oder, genauer gesagt, es gibt
   a) vollständige Zeilen (mindestens eine, nämlich die erste) und
   b) Zeilen, deren letztes Zeichen leer ist

Daraus ergibt sich folgende Schablone:

```
(define gt-liste-einlesen
  (lambda (gt-liste k)
      (cond
        (<Tabelle vollständig? .....)
        (else
         ..... )))
```

In beiden Fällen braucht man das folgende die Spaltenbreite k_{de} der GT-Tabelle, was wir mit

```
; k-de: n k --> spaltenbreite der GT-Liste
(: k-de (natural natural -> natural))
```

ermitteln, z.B.

```
(check-expect
 (k-de 21 4)
 6)
(check-expect
 (k-de 20 4)
 5)
```

In der Schablone

```
(define k-de
  (lambda (n k)
    (cond
      ((zero? (modulo n k)) .....)
      (else
      .....))))
```

Im ersten Fall, also wenn die Tabelle vollständig ist, muss n ist ganzzahlig durch k teilbar sein, also `(zero? (modulo n k))`, worauf `(quotient n k)` zurückgegeben wird, und im alternativen Fall `(+ 1 (quotient n k))`, weil eine zusätzliche, teilweise gefüllte Zeile notwendig wird. Insgesamt ergibt sich:

```
(define k-de
  (lambda (n k)
    (cond
      ((zero? (modulo n k)) (quotient n k))
      (else
       (+ 1 (quotient n k))))))
```

Jetzt können in der Schablone von `gt-liste-einlesen` den ersten Fall, nämlich der der vollständigen Tabelle durch `(zero? (modulo n k))` abprüfen und wie `wuerfeleinfach` mit dem Aufruf von `alle-einlesen` beantworten, allerdings jetzt statt $k$die neue Spaltenbreite der GT-Liste, nämlich `(k-de n k)` eingesetzt werde. Führen wir noch `n` als lokale Variable für die Länge der GT-Liste ein, ergibt sich:

```
(define gt-liste-einlesen
  (lambda (gt-liste k)
    (let ((n (length gt-liste)))
      (cond
        ((zero? (modulo n k)) (alle-einlesen gt-liste (k-de n k) 0))
        (else
         (append
           <vollständige Zeilen>
           <Zeilen mit leerem Zeichen am Ende> ))))))
```

Man sieht an dieser Schablone, dass man sich im zweiten Fall die unvollständige GT-Tabelle vorstellen kann als Zusammensetzung aus vollständigen Zeilen und unvollständigen Zeilen, d.h. mit einem leeren Zeichen am Ende, die jeweils als Listen modelliert werden. In beiden Fälle muss man wissen, welche und wieviele Zeichen der Zeichenkettenliste des GT jeweils einzulesen und in Listen zu transformieren sind.
Dafür entwickeln wir die Funktion

```
; teil-liste: liste anzahl zähler --> liste
(: teil-liste ((list %a) natural natural -> (list %b)))
```

die ersten `anzahl` Elemente einer Liste wegläßt, also z.B.

```
(check-expect
 (teil-liste
  (list "I" "R" "I" "C" "R" "E" "N" "M" "K" "H" "E"
        "F" "A" "M" "T" "U" "O" "T" "A" "F" "D") 6 0)
 (list "N" "M" "K" "H" "E" "F" "A" "M" "T" "U" "O" "T" "A" "F" "D"))
```

Das bedeutet in unserem Beispiel, dass die ersten 6 Elemente, die in eine vollständige Zeile eingelesen werden, weggelassen werden und eine Liste mit den restlichen $21 - 6 = 15$ Elementen, die in Zeilen mit leerem Zeichen am Ende eingelesen werden, übrig bleibt.
Es wird also eine Liste durchlaufen und ein Zähler ab 0 bei jedem Selbstaufruf hochgezählt: wenn der Zähler einen bestimmten Wert hat, nämlich ab `stelle`, wird ein Element der neuen Liste hinzugefügt, ansonsten erfolgt ein weiterer Selbstaufruf:

```
(define teil-liste
  (lambda (liste stelle zaehler)
    (cond
      ((empty? liste) empty)
      ((>= zaehler stelle)
       ..... (teil-liste (rest liste) ..... (+ zaehler 1))))
```

```
      (else
       (teil-liste (rest liste) ..... (+ zaehler 1))))))
```

Da ab `stelle` der neuen Liste das nächste Element hinzugefügt wird, wird die erste Ellipse durch `(cons (first liste)` gefüllt und der Parameter `stelle` bleibt unverändert beim Selbstruf, der den Rest der neuen Liste liefert. Im anderen Fall, also solange `(< zaehler stelle)`, ruft sich die Funktion selbst mit unverändertem `stelle` auf:

```
(define teil-liste
  (lambda (liste stelle zaehler)
    (cond
      ((empty? liste) empty)
      ((>= zaehler stelle)
       (cons (first liste) (teil-liste (rest liste) stelle (+ zaehler 1))))
      (else
       (teil-liste (rest liste) stelle (+ zaehler 1))))))
```

Um nun die Anzahl der Zeichen der GT-Liste, die in vollständige Zeilen eingelesen werden, zu ermitteln, gehen wir folgendermaßen vor: die Anzahl der vollständigen Zeilen, ist offensichtlich `(modulo n k)` und eine vollständige Zeile besteht - wie oben schon erläutert - aus `(k-de n k)` Zeichen, was der Spaltenbreite der GT-Tabelle entspricht. Somit ergeben sich insgesamt `(* (k-de n k) (modulo n k))` Zeichen. Umgekehrt bedeutet das, dass `(- n (* (k-de n k) (modulo n k)))` aus der GT-Liste entfernt werden müssen. Da diese Zeichen aber nicht am Anfang der Liste stehen, muss diese umgedreht werden, was insgesamt den Aufruf

```
(teil-liste (reverse gt-liste) (- n (* (k-de n k) (modulo n k)))0)
```

ergibt. Im vorliegenden Beispiel erhält man folgenden Testfall:

```
(check-expect
 (teil-liste
  (reverse (list "I" "R" "I" "C" "R" "E" "N" "M" "K" "H" "E" "F"
                 "A" "M" "T" "U" "O" "T" "A" "F" "D")))
  (- 21 (* (k-de 21 4) (modulo 21 4))) 0)
 (list "E" "R" "C" "I" "R" "I"))
```

Das bedeutet: Die GT-Liste für vollständige Zeilen enthält 6 Zeichen zum Einlesen, was eine vollständige Zeile in der GT-Tabelle ergeben wird, allerdings in umgekehrter Reihenfolge, was sich durch `reverse` beheben läßt. Jetzt kann mittels `alle-einlesen` die Liste der vollständigen Zeilen ermittelt werden, indem man als zweiten Paramter die Spaltenbreite `(k-de n k)` einsetzt. Die aktualisierte Funktionsschablone lautet:

```
(define gt-liste-einlesen
  (lambda (gt-liste k)
    (let ((n (length gt-liste)))
      (cond
        ((zero? (modulo n k)) (alle-einlesen gt-liste (k-de n k) 0))
        (else
         (append
          (alle-einlesen
           (reverse (teil-liste (reverse gt-liste) (- n (* (k-de n k) (modulo n k)))0))
           (k-de n k) 0)
          <Zeilen mit leerem Zeichen am Ende>))))))
```

Für das Einlesen der unvollständigen Zeilen muss `alle-einlesen` abgeändert werden zu

```
; alle-einlesen-mit-lz: GT-liste spaltenbreite zähler --> tabelle
(: alle-einlesen-mit-lz ((list string) natural natural -> (list (list string))))
```

Die Änderungen finden sich in den Elllipsen folgender Schablone:

```
(define alle-einlesen-mit-lz
  (lambda (gt-liste k y)
    (cond
      ((empty? gt-liste) empty)
      ((zero? (modulo y .....))
```

```
       (cons
        (..... (zeile-einlesen gt-liste ..... 0) .....)
        (alle-einlesen-mit-lz (rest gt-liste) k (+ y 1))))
      (else (alle-einlesen-mit-lz (rest gt-liste) k (+ y 1) )))))
```

Da jetzt jeweils nicht k, sondern nur noch $k-1$ Zeichen aus der GT-liste eingelesen, steht in der ersten Ellipse jetzt `(- k 1)`.
Wenn bei einem Selbstaufruf eine neue Zeile für die GT-Tabelle mit `cons` erzeugt wird, muss `zeile-einlesen` nur für `(- k 1)` Zeichen aufgerufen werden. Die so gelieferte Liste wird mittles `append` mit `(list "")` zu neuen Zeile mit leerem Zeichen am Ende verknüpft.
Insgesamt ergibt sich

```
(define alle-einlesen-mit-lz
  (lambda (gt-liste k y)
    (cond
      ((empty? gt-liste) empty)
      ((zero? (modulo y (- k 1)))
       (cons
        (append (zeile-einlesen gt-liste (- k 1) 0) (list ""))
        (alle-einlesen-mit-lz (rest gt-liste) k (+ y 1))))
      (else (alle-einlesen-mit-lz (rest gt-liste) k (+ y 1) )))))
```

Aufgerufen wird `alle-einlesen-mit-lz` innerhalb von `gt-liste-einlesen` mit der Teilliste derjenigen Zeichen aus der GT-Liste, die noch nicht für die vollständigen Zeilen verbrauchen wurden: Es handelt sich - wie oben dargelegt - dabei um `(* (k-de n k) (modulo n k))` Elemente, die mittels `teil-liste` aus `gt-liste` als Liste geliefert werden, insgesamt erhält man:

```
(define gt-liste-einlesen
  (lambda (gt-liste k)
    (let ((n (length gt-liste)))
      (cond
        ((zero? (modulo n k)) (alle-einlesen gt-liste (k-de n k) 0))
        (else
         (append
          (alle-einlesen
           (reverse
            (teil-liste
             (reverse gt-liste) (- n (* (k-de n k) (modulo n k)))0))
           (k-de n k) 0)
          (alle-einlesen-mit-lz
           (teil-liste gt-liste (* (k-de n k) (modulo n k)) 0)
           (k-de n k) 0)))))))
```

Damit ist Entschlüsselungsfunktion `de-wuerfeleinfach` komplett:

```
(define de-wuerfeleinfach
  (lambda (geheimtext k)
    (apply string-append
           (map strings-list->string
                (transponiere
                 (gt-liste-einlesen
                  (string->strings-list (bereinige-klartext geheimtext)) k))))))
```

Übungen

1. Ändere `wuerfeleinfach` so, dass
   a) die Spalten von unten nach oben ausgelesen werden
   b) die Tabelle von rechts nach links ausgelesen wird

4.3 Spaltentransposition mit Permutation

Hierbei handelt es sich um eine Variante von `wuerfeleinfach`: Statt die KT-Tabelle von links nach rechts, also in der Reihenfolge der Spaltennummern $1, 2, ..., k$ auszulesen, wird sie in der Reihenfolge

einer vorher festgelegten Permutation der Spaltennummern ausgelesen: im Falle $k = 4$ werden die Spalten nicht in der Reihenfolge $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ ausgelesen, sondern etwa $2 \rightarrow 3 \rightarrow 4 \rightarrow 1$.
Eine solche Permutation der Spalten hat zur Folge, dass bei einer anschließenden algebraischen Transposition der KT-Tabelle die Zeilen bei der so entstandenen GT-Tabelle entsprechend permutiert sind. Das bedeutet, dass eine spaltenweise Permutation SP der KT-Tabelle mit anschließender algebraischer Transposition T und eine algebraische Transposition T der KT-Tabelle mit anschließender zeilenweiser Permutation ZP äquivalent sind:

$$T(SP(KT\text{-}Tabelle)) \cong (ZP(T(KT\text{-}Tabelle))$$

Dabei ist offensichtlich, dass sich eine Funktion

```
; permutiere: liste permutation --> liste
(: permutiere ((list %a) (list natural) -> (list %b)))
```

für eine Zeilenpermutation leichter realisieren läßt als für eine Spaltenpermutation, weil in der algebraisch transponierten Matrix die Zeilen durch Listen repräsentiert werden, was in der KT-Tabelle für die Spalten nicht gilt.
Insgesamt folgt daraus, dass die gesuchte Verschlüsselungsfunktion

```
; spalten-perm: KT Permutation --> GT
(: spalten-perm (string string -> string))
```

sich von `wuerfeleinfach` nur dadurch unterscheidet, dass durch Transposition der KT-Tabelle entstandene Tabelle mit Hilfe von `permutiere` permutiert werden muss:

```
(define spalten-perm
  (lambda (klartext .....)
    ......
      (apply string-append
             (map strings-list->string
                  (permutiere
                   (transponiere (kt->tabelle (bereinige-klartext klartext) .....))
                   .....))))))
```

Zunächst konstruieren wir `permutiere`: Modelliert man die Permutation als Liste der Zeilennummern, etwa `(list 2 3 4 1)`, wird man z.B.

```
(check-expect
 (permutiere
  (list
   (list "I" "R" "I" "C" "R" "E")
   (list "N" "M" "K" "H" "E" "")
   (list "F" "A" "M" "T" "U" "")
   (list "O" "T" "A" "F" "D" "")) (list 2 3 4 1))
 (list
  (list "N" "M" "K" "H" "E" "")
  (list "F" "A" "M" "T" "U" "")
  (list "O" "T" "A" "F" "D" "")
  (list "I" "R" "I" "C" "R" "E")))
```

erwarten. Die Funktion

```
(define permutiere
  (lambda (liste permutationlise)
        ......))
```

durchläuft eine List, die Permutationsliste, und konstruiert dabe eine neue Liste, die GT-Tabelle:

```
(define permutiere
  (lambda (liste permutationlise)
    (cond
      ((empty? permutationlise) empty)
      (else
       (cons ..... (first permutationlise) .......
             (permutiere liste (rest permutationlise)))))))
```

Bei jedem Aufruf liefert `(first permutationlise)` die Nummer der entsprechenden Zeile, die mit der eingebauten Funktion `(list-ref liste stelle` aus der ursprünglichen Liste ausgewählt wird, allerdings muss der Wert um 1 erniedrigt werden, da der Index `stelle` bei 0 beginnt. Wir erhalten so

```
(define permutiere
  (lambda (liste permutationlise)
    (cond
      ((empty? permutationlise) empty)
      (else
       (cons (list-ref liste (- (first permutationlise) 1))
             (permutiere liste (rest permutationlise)))))))
```

Während hier die Permutation als zweiter Parameter in Form einer Liste von ganzen Zahlen vorliegen muss, ist es beim Hauptprogramm `spalten-perm` sicher einfacher, die Permutation als zweiten Parameter in Form einer Zeichenkette einzugeben, also z.B. `"2341"` statt `(list 2 3 4 1)` . Die Bereitstellung als Liste kann durch die lokale Definition

```
(let ((perm-liste (map string->number (string->strings-list permutation)))) ....)
```

erfolgen.
Wer hier den bisherigen Parameter k als Spaltenbreite im Funktionsvertrag vermißt hat, wird schnell feststellen, dass sich dieser jetzt erübrigt, da die die Länge der Permutationsliste, also z.B. von `(list 2 3 4 1)` bzw. die Länge der Permutation `"2341"` die Spaltenbreite liefert, also in diesem Fall $k = 4$.
Somit können wir in dem o.a. Gerüst von `spalten-perm` die ersten beiden Ellipsen füllen:

```
(define spalten-perm
  (lambda (klartext permutation)
    (let ((perm-liste (map string->number (string->strings-list permutation))))
      (apply string-append
             (map strings-list->string
                  (permutiere
                   (transponiere (kt->tabelle (bereinige-klartext klartext) .....))
                   .....))))))
```

Die verbleibenden Ellipsen entsprechen den noch fehlenden Parametern bei `kt->tabelle` und `permutiere`: Wie bereits gesagt, wird bei der ersten Ellipse statt k der Wert von `(length permliste)` genommen und bei `permutiere` die Zahlenliste `perm-liste`.
Ein Ergebniss wie z.B.

```
(check-expect
 (spalten-perm "INFORMATIKMACHTFREUDE" "2341")
 "NMKHEFAMTUOTAFDIRICRE")
```

liefert dann

```
(define spalten-perm
  (lambda (klartext permutation)
    (let ((perm-liste (map string->number (string->strings-list permutation))))
      (apply string-append
             (map strings-list->string
                  (permutiere
                   (transponiere
                    (kt->tabelle
                     (bereinige-klartext klartext)
                     (length perm-liste)))
                   perm-liste))))))
```

Entschlüsselung

s. Übungen

Übungen

1. Ändere `spalten-perm` so, dass
   a) die Spalten von unten nach oben ausgelesen werden
2. Entwickle zu `spalten-perm` ein Dechiffrierprogramm

Literatur

[1] *Klaeren, Herbert* und *Sperber, Michael*, **Die Macht der Abstraktion** – Einführung in die Programmierung –, 1. Auflage, Teubner, 2007, ISBN-Nr.978-3-8351-0155-5

[2] *Felleisen, M.*, u.a., **How to Design Programs** – An Introduction to Programming and Computing –, MIIT Press, 2001, ISBN-Nr. 0-262-06218-6, online-Version (engl.): *http://www.htdp.org*