Institut für Informationssysteme
Fachgebiet Programmiersprachen und Übersetzer
Universität Hannover
Prof. Dr. Rainer Parchmann

Prüfungen Programmieren 1

in Scheme

WS 2002/03

Vielen Dank an Prof. Dr. Rainer Parchmann für die Aufgabenblätter

Jeder Student erhielt eines der Aufgabenblätter. Alle Teilaufgaben eines Blattes mussten gelöst werden um zu bestehen. Es gab nur *bestanden* oder *nicht bestanden*.

Für das Programmieren wurde *drscheme* genutzt mit *Language=Full Scheme*.

Aufgabe A

Universität Hannover
Institut für Informationssysteme
Fachgebiet Programmiersprachen und Übersetzer
Prof. Dr. R. Parchmann

Programmieren I
WS 2002/2003

Prüfungsaufgabe A vom 17.02.2003

Ist $n \in \mathbb{N}$ so heißt $M_n := 2^n - 1$ die *n-te Mersennsche Zahl*. Ist diese Zahl außerdem eine Primzahl, so spricht man von einer *Mersennschen Primzahl*. Es ist $M_1 = 1$, $M_2 = 3$, $M_3 = 7$, $M_4 = 15$ usw.
Der sogenannte *Lucas-Lehmer-Test* ermittelt zu gegebenen n, ob es sich bei der n-ten Mersennschen Zahl M_n um eine Primzahl handelt. Er beruht auf folgendem Satz:

> Sei $n \in \mathbb{N}$, $n \geq 3$. Weiter sei die Folge $a_0, a_1, \ldots$ rekursiv definiert durch $a_0 := 4$ und $a_i := a_{i-1}^2 - 2$ für $i > 0$. Dann gilt: M_n ist genau dann prim, wenn $a_{n-2} \bmod M_n = 0$ ist.

Schreiben Sie eine Scheme-Prozedur `lucas-lehmer` mit einem Parameter `n`, die für ein gegebenes $n \in \mathbb{N}$ testet, ob M_n prim ist.
Beispiele:

```
(lucas-lehmer 1)  ==> #f
(lucas-lehmer 2)  ==> #t
(lucas-lehmer 3)  ==> #t
(lucas-lehmer 5)  ==> #t
(lucas-lehmer 7)  ==> #t
(lucas-lehmer 11) ==> #f
(lucas-lehmer 13) ==> #t
(lucas-lehmer 17) ==> #t
(lucas-lehmer 19) ==> #t
(lucas-lehmer 23) ==> #f
```

Gehen Sie bei der Implementierung wie folgt vor:

- Berechnen Sie zunächst M_n und merken Sie sich diesen Wert in einer lokalen Variablen. Benutzen Sie die vordefinierte Scheme-Prozedur `expt`. (Beispiel: `(expt 2 8)` liefert den Wert `256`)
- Schreiben Sie nun eine rekursive lokale Hilfsprozedur `folge`, die zu gegebenem i (`i` ist der Parameter) das Folgenglied a_i berechnet. Die Berechnungsregel ist dem Satz zu Beginn der Aufgabenstellung zu entnehmen: $a_0 := 4$ und $a_i := a_{i-1}^2 - 2$, $i \in \mathbb{N}$. Beispiel:

  ```
  (folge 1)  ==> 14
  (folge 2)  ==> 194
  (folge 3)  ==> 37634
  (folge 4)  ==> 1416317954
  ```

- Modifizieren Sie nun diese Hilfsprozedur, um statt des i-ten Folgengliedes das i-te Folgenglied modulo M_n zu erhalten. Nutzen Sie für die Moduloberechnung die Prozedur `remainder`. (Beispiel: `(remainder 8 3)` liefert 2 (das entspricht 8 mod 3)). Verwenden Sie: Es ist $a_i \bmod M_n = (a_{i-1} \bmod M_n)^2 - 2 \bmod M_n$ für $i > 0$.
- In der Prozedur `lucas-lehmer` müssen die Fälle $n = 1$ und $n = 2$ gesondert behandelt werden. Ist dagegen $n \geq 3$, so benutzen Sie die Hilfsprozedur um $a_{n-2} \bmod M_n$ zu berechnen. Entsprechend dem oben angegebenen Satz bestimmen Sie dann das Ergebnis.

- **Die Bearbeitungszeit beträgt 45 Minuten.**
- **Bei Fragen wenden Sie sich bitte an einen Betreuer.**
- **Viel Erfolg!**

Lösung A

Prozedur `M` mit Parameter `n`, die die *Mersennschen Zahl* berechnet:

```
(define (M n)
 (- (expt 2 n) 1))
```

Prozedur `folge` mit Parameter `i`, die die Lucas-Folge berechnet:

```
(define (folge i)
 (if (= i 0)
      4
      (- (expt (folge (- i 1)) 2) 2) ))
```

Prozedur `lucas-lehmer`, mit Parameter `n` und implizierter Hilfsprozedur `(folge i)` , die überprüft, ob eine gegebene Zahl n prim ist:

```
(define (lucas-lehmer n)
  (let ((Mn (M n)))
   (define (folge i)
     (if (= i 0)
         4
         (- (expt (remainder (folge (- i 1)) Mn) 2) 2) ))
 (cond ((= n 1) #f)
       ((= n 2) #t)
       (else (if (= 0 (remainder (folge (- n 2)) Mn)) #t #f)) )))
```

Lösung von vier

Aufgabe B

Universität Hannover
Institut für Informationssysteme
Fachgebiet Programmiersprachen und Übersetzer
Prof. Dr. R. Parchmann

Programmieren I
WS 2002/2003

Prüfungsaufgabe B vom 17.02.2003

Die **Glücklichen Zahlen** erhält man ähnlich wie die Primzahlen mit einem Siebverfahren. Man beginnt mit den natürlichen Zahlen größer gleich 2, d.h. der Folge: 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, ... Das erste Element dieser Folge, also die 2, ist die erste *glückliche Zahl*.
Man entfernt nun aus dieser Folge die 2 (also das erste Element) und erhält so eine neue Folge 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, ... Dann entfernt man aus dieser Folge jedes zweite Element und erhält die neue Folge: $3, 5, 7, 9, 11, 13, \ldots$.
Die nächste glückliche Zahl ist das erste Element dieser Folge, also die 3. Diese entfernt man aus der Folge, streicht danach nun jede dritte Zahl der Folge und das neue erste Element ist die dritte glückliche Zahl (die neue Folge ist also $5, 7, 11, 13, \ldots$).
Allgemein: Ist n eine glückliche Zahl, so steht sie zu irgendeinem Zeitpunkt am Anfang der Folge. Man entfernt dieses n (das erste Element) aus der Folge und streicht aus der verbleibenden Folge jedes n-te Element. Das danach vorne stehende Element ist wiederum eine glückliche Zahl.
Die Folge der Glücklichen Zahlen beginnt also mit: $2, 3, 5, 7, 11, 13, 17, 23, 25, 29, \ldots$.
Ziel ist es nun, die Berechnung dieser Zahlenfolge in Scheme zu implementieren. Gehen Sie dabei wie folgt vor:

- Erstellen Sie eine Hilfsfunktion, die aus einer gegebenen Liste `li` ab einem Index k jedes n-te Element entfernt, ohne dabei die Reihenfolge der Elemente zu verändern. (Um Missverständnisse zu vermeiden: Es ist nicht notwendig die gegebene Liste zu verändern, es reicht, aus den Elementen der gegebenen Liste eine neue Ergebnisliste zusammenzustellen.) Es sollen also die Elemente mit den Indizes k, $k+n$, $k+2n$, $k+3n$, ... entfernt werden, wobei das erste Element den Index 1 habe.

  Vervollständigen Sie also `(define (entferne-jedes-n-te-hilf li n k) ...)`. Dabei ist es sinnvoll, folgende Fälle zu unterscheiden:

  - Der Fall, dass die Liste `li` leer ist, bedarf keiner näheren Erklärung.
  - Im Fall $k = 1$ entfernen Sie das erste Listenelement und rufen die Hilfsfunktion rekursiv mit der restlichen Liste, dem unveränderten n und dem Wert von n als neues k auf.
  - Andernfalls fügen Sie das erste Element von `li` zur Ergebnisliste hinzu und rufen die Hilfsfunktion rekursiv mit der restlichen Liste, dem unveränderten n und dem um 1 erniedrigten k auf.

- Schreiben Sie nun die Funktion `entferne-jedes-n-te` mit den beiden Parametern `li` und n, die aus der Liste `li` jedes n-te Element entfernen soll. Hierzu muss man nun nur noch `entferne-jedes-n-te-hilf` aufrufen. (Als Anfangswert für k muss offenbar n gewählt werden.) Anwendungsbeispiel:

```
(entferne-jedes-n-te '() 17)                          ==> ()
(entferne-jedes-n-te '(3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20)  2)
                                                      ==> (3 5 7 9 11 13 15 17 19)
(entferne-jedes-n-te '(5 7 11 13 17 19) 3)              ==> (5 7 13 17)
```

- Schreiben Sie nun die Funktion `sieve`, die einen Parameter `li` (erneut eine Liste) besitzt. Diese interpretiert die Eingabe stets als Anfangsstück einer der Folgen, die bei der Berechnung der Glücklichen Zahlen auftreten und filtert die in ihm enthaltenen Glücklichen Zahlen heraus und gibt sie als Liste zurück. Beispiele:

```
(sieve '(2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20))   ==>  (2 3 5 7 11 13 17)
(sieve '(3 5 7 9 11 13 15 17 19))                             ==>  (3 5 7 11 13 17)
(sieve '(5 7 11 13 17 19))                                    ==>  (5 7 11 13 17)
```

Gehen Sie bei der Implementierung wie folgt vor:

- Ist `li` die leere Liste, dann geben Sie die leere Liste als Ergebnis zurück.
- Andernfalls ist das erste Element eine Glückliche Zahl und wird somit in die Ergebnisliste aufgenommen. Sie rufen nun `entferne-jedes-n-te` mit der restlichen Liste und dieser Glücklichen Zahl als Argumente auf und verwenden die so erhaltene Liste beim rekursiven Aufruf von `sieve`.

- Nun erstellen Sie noch eine Scheme-Funktion `glueckliche-zahlen-bis-n`, die einen Parameter n besitzt und eine Liste aller Glücklichen Zahlen bis n zurückgibt. Die folgende Scheme-Funktion wird Ihnen dabei behilflich sein:

```
(define (interval i j)
    (if (> i j)
        '()
        (cons i (interval (+ i 1) j))))
```

Anwendungsbeispiele:

```
(glueckliche-zahlen-bis-n 20)  ==>  (2 3 5 7 11 13 17)
(glueckliche-zahlen-bis-n 33)  ==>  (2 3 5 7 11 13 17 23 25 29)
(glueckliche-zahlen-bis-n 100) ==>  (2 3 5 7 11 13 17 23 25 29 37 41
                                     43 47 53 61 67 71 77 83 89 91 97)
```

- **Die Bearbeitungszeit beträgt 45 Minuten.**
- **Bei Fragen wenden Sie sich bitte an einen Betreuer.**
- **Viel Erfolg!**

Lösung B

```
(define (entferne-jedes-n-te-ab-k li n k)
  (cond ((null? li) '() )
        ((= k 1) (entferne-jedes-n-te-ab-k (cdr li) n n))
        (else (cons (car li) (entferne-jedes-n-te-ab-k (cdr li) n (- k 1)) ))))

(define (entferne-jedes-n-te li n)
  (entferne-jedes-n-te-ab-k li n n))

(define (sieve li)
  (if (null? li)
      '()
      (cons (car li) (sieve (entferne-jedes-n-te (cdr li) (car li))) )))

(define (interval j i)
  (if (> j i)
      '()
      (cons j (interval (+ j 1) i))))

(define (glueckliche-zahlen-bis-n n)
  (sieve (interval 2 n)))

(glueckliche-zahlen-bis-n 100)
```

Lösung von metalhen

Aufgabe C

Universität Hannover
Institut für Informationssysteme
Fachgebiet Programmiersprachen und Übersetzer
Prof. Dr. R. Parchmann

Programmieren I
WS 2002/2003

Prüfungsaufgabe C vom 18.02.2003

Bei der Speicherung oder Übertragung einer Folge von Binärziffern (also einer Folge von 0en und 1en) werden die Daten so codiert, dass später möglichst eine einwandfreie Rekonstruktion (auch bei eventuellen Störungen) möglich wird. Oft wird eine sog. MFM-Codierung benutzt. Hierbei wird zwischen je zwei benachbarte Bits der Originalfolge ein weiteres sog. Taktbit eingefügt. Dieses Bit ist genau dann 1, wenn die benachbarten Bits beide 0 sind. Vor das erste und hinter das letzte Element der Originalfolge wird kein Taktbit eingefügt. Die Folge 10011 wird also als 1<u>0</u>0<u>1</u>0<u>0</u>1<u>0</u>1 codiert (Die Taktbits sind unterstrichen.)

- Schreiben Sie eine Prozedur `codiere-mfm` mit einem Parameter `li`, die zu der als Liste `li` gegebenen 0-1-Folge die MFM-Codierung berechnet. Es könnte hilfreich sein, eine Hilfsprozedur zu schreiben, die als zusätzlichen Parameter das zuletzt behandelte Zeichen der Originalfolge erhält. Anwendungsbeispiele:

```
(codiere-mfm '())           ==> ()
(codiere-mfm '(1))          ==> (1)
(codiere-mfm '(1 0 0 1 1))  ==> (1 0 0 1 0 0 1 0 1)
```

- Schreiben Sie die Prozedur `decodiere-mfm` die als Parameter eine MFM-codierte Folge erhält und daraus die uncodierte Folge rekonstruiert. (Sie müssen lediglich jedes zweite Listenelement entfernen.) Anwendungsbeispiele:

```
(decodiere-mfm '(1 0 0 1 0 0 1 0 1))  ==> (1 0 0 1 1)
(decodiere-mfm '(1 0 0))              ==> (1 0)
(decodiere-mfm '(1))                  ==> (1)
(decodiere-mfm '())                   ==> ()
```

- Schreiben Sie nun eine Scheme-Prozedur `besseres-decodiere-mfm`, die ebenfalls eine MFM-codierte Liste als Eingabe (der einzige Parameter) erhält und daraus die Originalliste rekonstruiert, aber zusätzlich überprüft, ob die Eingabe tatsächlich als MFM-Codierung vorliegt. (Sie können aber trotzdem davon ausgehen, dass die Liste nur 0en und 1en als Elemente besitzt).

  Ist dies nicht der Fall, so wird das Symbol `fehler` zurückgeliefert. Ein Fehler liegt also in folgenden Fällen vor:

  – Die Länge der Liste ist gerade und ungleich 0.
  – Eines der Taktbits ist falsch gesetzt.

  Bei der Implementierung von `besseres-decodiere-mfm` gehen Sie wie folgt vor:

  – Falls die Länge der Liste gerade und ungleich 0 ist, geben Sie `fehler` aus.
  – Andernfalls: Decodieren Sie die Liste mit `decodiere-mfm`, speichern die daraus erhaltene Liste in einer lokalen Variablen und prüfen diese Liste auf inhaltliche Gleichheit mit der Originaleingabe. Sind die beiden Listen gleich, geben Sie die gespeicherte Liste, ansonsten das Symbol `fehler` zurück.

  Anwendungsbeispiele:

```
(besseres-decodiere-mfm '(1 0 0 1 0 0 1 0 1))  ==> (1 0 0 1 1)
(besseres-decodiere-mfm '(1 0 0 0))            ==> fehler
(besseres-decodiere-mfm '(1 1 0))              ==> fehler
(besseres-decodiere-mfm '(1))                  ==> (1)
```

- **Die Bearbeitungszeit beträgt 45 Minuten.**
- **Bei Fragen wenden Sie sich bitte an einen Betreuer.**
- **Viel Erfolg!**

Lösung C

```
(define (codiere-mfm li)
  (define (codiere li a nli)
    (if (null? li)
        (append nli (list a))
        (cond
          ((and (= a 0)
                (= a (car li))) (codiere (cdr li) (car li) (append nli (list a 1))))
          (else (codiere (cdr li) (car li) (append nli (list a 0)))))))
 (if (null? li) () (codiere (cdr li) (car li) ())))

(define (decodiere-mfm li)
  (define (decodiere li nli)
    (if (null? li)
        nli
        (decodiere (cddr li) (append nli (list (car (cdr li)))))))
 (if (null? li)
     ()
     (decodiere (cdr li) (list (car li)))))

(define (besseres-decodiere-mfm li)
  (if (and (even? (length li))
           (> (length li)  0))
      (display "Fehler!")
      (let ((nli (decodiere-mfm li)))
        (if (equal? (codiere-mfm nli) li)
            nli
            (display "Fehler!")))))


(newline)
(display "codiere-mfm")     (codiere-mfm '(1 0 0 1 1))
(codiere-mfm '(1))          (codiere-mfm '())
(newline)
(display "decodiere-mfm") (decodiere-mfm (codiere-mfm '(1 0 0 1 1)))
(decodiere-mfm '(1))      (decodiere-mfm '())
(newline)
(display "besseres-decodiere")       (besseres-decodiere-mfm '(1 0 0 1 0 0 1 0 1))
(besseres-decodiere-mfm '(1 0 0 0))  (besseres-decodiere-mfm '(1 0))
(besseres-decodiere-mfm '(1))        (besseres-decodiere-mfm '())
```

Lösung von Cipher

Aufgabe D

Universität Hannover
Institut für Informationssysteme
Fachgebiet Programmiersprachen und Übersetzer
Prof. Dr. R. Parchmann

Programmieren I
WS 2002/2003

Prüfungsaufgabe D vom 18.02.2003

Bei der Speicherung oder Übertragung von Daten werden die einzelnen Datenelemente durch eine Folge von Bits (also einer Folge von 0en und 1en) so codiert, dass später möglichst eine einwandfreie Decodierung auch bei eventuellen Störungen möglich wird. Die Störungen bestehen darin, dass statt einer gesendeten 0 eine 1 oder statt einer gesendeten 1 eine 0 empfangen wird. Im optimalen Fall kann der Empfänger aufgrund der Eigenschaften der Codierung auch dann noch die ursprünglichen Daten rekonstruieren, wenn Übertragungsfehler einzelne Bits verändern haben. Dies ist natürlich nur möglich, wenn die Anzahl der veränderten Bits eine gewisse Schranke nicht überschreitet.
In diesem Zusammenhang benutzt man oft den Begriff der *Hamming-Distanz*, die angibt, in wievielen Bits sich zwei Binärfolgen gleicher Länge unterscheiden.
Beispiel: (1 1 0 1 0) unterscheidet sich von (1 1 1 1 0) in genau einer Stelle, also haben die beiden Folgen eine Haming-Distanz von 1. Die Hamming-Distanz der beiden Binärfolgen (0 1 0 0 1) und (0 0 0 1 1) ist dagegen 2.

- Schreiben Sie eine Prozedur `hamming`, die die Hamming-Distanz zweier Codewörter gleicher Länge berechnet. Sie können davon ausgehen, dass beide Parameter Listen von 0en und 1en gleicher Länge sind.

  ```
  (hamming '(0 1 0 0 1) '(0 0 0 1 1))  ==> 2
  (hamming '(1 1 0 1 0) '(1 1 1 1 0))  ==> 1
  ```

Eine mögliche Strategie bei der Decodierung besteht nun darin, die empfangene Binärfolge mit allen möglichen Codewörtern zu vergleichen und das Codewort auszuwählen, das mit der empfangenen Binärfolge die kleinste Hamming-Distanz hat. Man betrachte etwa die folgende Liste von Codewörtern

```
(define cwliste-bsp '((0 0 0 0 0) (0 0 1 1 1) (1 1 1 0 0) (1 1 0 1 1)))
```

und interpretiere hierbei das Codewort (0 0 0 0 0) als Codierung der Zahl 0, das Codewort (0 0 1 1 1) als Codierung der Zahl 1, u. s. w. bis zum Codewort (1 1 0 1 1) als Codierung der Zahl 3. Empfängt man jetzt die Binärfolge (0 1 0 1 1), so berechnet man, dass die Hamming-Distanz zur Codierung der 0 gleich 3, zur Codierung der 1 gleich 2, zur Codierung der 2 gleich 4 und zur Codierung der 3 gleich 1 ist. Also legt man fest, dass der Sender die Zahl 3 gesendet hat.
Eine solche Liste von Codierungen nennen wir im folgenden *Codewortliste*. Das i-te Element stellt hierbei die Codierung der Zahl i dar.

- Schreiben Sie eine Scheme-Prozedur `make-codierer` mit einem Parameter `cwliste`, die als Ergebnis eine Codierungsfunktion liefert, die einer Zahl i das i-te Element der Codewortliste `cwliste` zuordnet. Anwendungsbeispiel:

  ```
  (define codierer (make-codierer cwliste-bsp))

  (codierer 0)  ==>  (0 0 0 0 0)
  (codierer 1)  ==>  (0 0 1 1 1)
  (codierer 3)  ==>  (1 1 0 1 1)
  ```

  Implementierungshinweis:

  - Sie können die Scheme-Prozedur `list-ref` benutzen. Anwendungsbeispiele:

    ```
    (list-ref '(a b c) 0) ==> a
    (list-ref '(a b c) 2) ==> c
    ```

- Schreiben Sie eine Scheme-Prozedur `decodiere` mit den Parametern `cw`, `cwliste` und `max-distanz`. Wenn die *Hamming-Distanz* des Codewortes `cw` mit jedem der Codewörter in `cwliste` größer als `max-distanz` ist, dann soll als Ergebnis das Symbol `fehler` geliefert werden (der Übertragungsfehler ist also zu groß als dass er sinnvoll korrigiert werden könnte).

  Andernfalls soll der Index desjenigen Codewortes aus `cwliste` zurückgeliefert werden, das die minimale Hamming-Distanz zu `cw` besitzt. (Wenn mehrere Listenelemente in Frage kommen, ist es egal, welcher der zugehörigen Indizes zurückgeliefert wird.)

- Schreiben Sie dazu eine lokale Hilfsprozedur, die die folgenden Parameter besitzt:
  - `index-akt`: der Index des Wortes der Codewortliste, das bislang die geringste Hamming-Distanz aufgewiesen hat.
  - `distanz-akt`: die bislang geringste gefundene Hamming-Distanz.
  - `rest-cwliste`: die noch zu untersuchende Teilliste der Codewortliste.
  - `index`: ein Zähler, der mitzählt, wieviele Elemente schon untersucht wurden.

  Die Hilfsfunktion muss folgende Fälle unterscheiden:
  - ist die noch zu untersuchende Liste leer, wird der bisher gefundene Index mit der kleinsten Hamming-Distanz zurückgeliefert.
  - Besitzt das erste Wort der Codewortliste eine geringere Hamming-Distanz als alle zuvor untersuchten Worte, wird die Hilfsfunktion rekursiv mit der Restliste und dem Index des aktuellen Wortes sowie der neu gefundenen minimalen Hamming-Distanz aufgerufen.
  - Andernfalls hat das erste Wort der Codewortliste keine geringere Hamming-Distanz als zuvor gefunden wurde und die Hilfsfunktion muss rekursiv aufgerufen werden um noch die Restliste zu untersuchen.
- Zur Implementation von `decodiere` muss diese Hilfsfunktion nun nur noch mit den richtigen Werten aufgerufen werden.

  Anwendungsbeispiele:

```
(decodiere '(0 0 0 0 0) cwliste-bsp 1)  ==>  0
(decodiere '(1 1 1 0 0) cwliste-bsp 1)  ==>  2
(decodiere '(1 0 0 0 0) cwliste-bsp 1)  ==>  0
(decodiere '(0 1 1 1 0) cwliste-bsp 1)  ==>  fehler
```

- **Die Bearbeitungszeit beträgt 45 Minuten.**
- **Bei Fragen wenden Sie sich bitte an einen Betreuer.**
- **Viel Erfolg!**

Lösung D

```
(define (hamming liste1 liste2)
  (define (ham-iter liste1 liste2 Ergebnis)
    (if (null? liste1)
        Ergebnis
        (if (= (car liste1) (car liste2))
            (ham-iter (cdr liste1) (cdr liste2) Ergebnis)
            (ham-iter (cdr liste1) (cdr liste2) (+ Ergebnis 1)))))
 (ham-iter liste1 liste2 0))


(define cwliste-bsp '((0 0 0 0 0) (0 0 1 1 1) (1 1 1 0 0) (1 1 0 1 1)))

(define (make-codierer cwliste-bsp)
  (lambda (n) (list-ref cwliste-bsp n)))

(define codierer (make-codierer cwliste-bsp))


(define (decodiere cw cwliste max-distanz)
  (define (CodeTest index-akt distanz-akt rest-cwliste index)
    (if (null? rest-cwliste)
        index-akt
        (if (< (hamming cw (car rest-cwliste)) distanz-akt)
            (CodeTest index (hamming cw (car rest-cwliste))
                                  (cdr rest-cwliste) (+ index 1))
            (CodeTest index-akt distanz-akt (cdr rest-cwliste) (+ index 1)))))
 (CodeTest 'fehler! (+ max-distanz 1) cwliste 0))
```

Lösung von Brainbug