Ludwig-Maximilians-Universität München
Institut für Informatik
Prof. Dr. F. Kröger, Dr. M. Hölzl, F. Hacklinger

Übungen zu Informatik I (Lösungsvorschlag)

Aufgabe 2-1 **Wechselgeld** (keine Abgabe)

In dieser Lösung werden die auszugebenden Münzen als eine Folge von den Münzwerten wiedergegeben. Dementsprechend würden drei 2-Euro-Münzen, und dann eine 1-Euro-Münze durch [2],[2],[2],[1] wiedergegeben.

Die erste rekursive Lösungsvariante zählt in Zweierschritten vom Preis der Fahrkarte nach oben („Kassierervariante"), bis entweder die 100 erreicht ist (es muß nichts mehr getan werden) oder bis 99 (dann muß noch eine 1-Euro-Münze angefügt werden).

```
Wrek1 = function (preis : nat) folge[muenzwerten] :
  pre 1 ≤ preis ≤ 100

  result Folge von 1− und 2−Euro−Muenzwerten, so dass preis zusammen
  mit der Summe aller Muenzwerte 100 ergibt.
  if preis = 100 then []
  else if preis = 99 then [1]
  else [2] gefolgt von Wrek1(preis + 2)
```

Eine leicht abgewandelte Variante bestimmt erst das Restgeld und „gibt" solange Münzen aus, bis der Restgeldbetrag erreicht ist. Zur Realisierung wird eine Hilfsfunktion verwendet, die zu einem gegebenen Betrag eine Folge von 1- und 2-Euro-Münzen zurückgibt, so dass der Wert der Münzen dem Betrag entspricht.

```
Wrek2 = function (preis : nat) folge[muenzwerten] :
  pre 1 ≤ preis ≤ 100

  result Folge von 1− und 2−Euro−Muenzwerten, so dass preis zusammen
  mit der Summe aller Muenzwerte 100 ergibt.
  let
    restbetrag = 100 − preis
  in
    Muenzen(restbetrag)
  end

Muenzen = function (betrag : nat) folge[muenzwerten] :
  pre betrag ≥ 0

  result Folge von 1− und 2−Euro−Muenzen, so dass die Summe der
  Muenzwerte dem Argument betrag entspricht
  if betrag = 0 then []
  else if betrag = 1 then [1]
  else [2] gefolgt von Muenzen(betrag − 2)
```

Aufgabe 2-2 **Sitzordnung** (keine Abgabe)

Die gesuchte Funktion kann man durch Rekursion nach der Anzahl der Paare, also nach n, definieren. Zunächst ist klar, dass für 1 Paar genau zwei mögliche Reihenfolgen existieren. Man muss sich nun überlegen, wie man die Anzahl der möglichen geeigneten Sitzordnungen für n Paare auf die Anzahl der entsprechenden Sitzordnungen für $n-1$ Paare zurückführen kann.

Für eine feste Reihenfolge für $n-1$ Paare hat das n-te Paar (als Paar, also zunächst noch ohne Beachtung der Reihenfolge zwischen Mann und Frau) genau n Plätze zur Auswahl, nämlich etweder an einem Ende der Reihe (2 Möglichkeiten), oder irgendwo „in der Mitte“ ($n-2$ Möglichkeiten). Damit sind es insgesamt n Möglichkeiten. Wenn man jetzt auch noch beachtet, dass das Ehepaar untereinander auch noch je zwei mögliche Reihenfolgen hat, ergibt sich insgesamt, dass es für eine feste Reihenfolge von $n-1$ Paaren das n-te Ehepaar $2{\cdot}n$ Möglichkeiten hat, sich zu setzen. Diese Beobachtung führt direkt zur folgenden rekursiven Funktion:

sitzordnung = **function** (n:**nat**) **nat**:
pre n $\geq$ 1

result Anzahl der moeglichen verschiedenen Sitzordnungen gemaess Angabe
  if n = 1 **then** 2
  else 2∗n∗sitzordnung(n−1)

Aufgabe 2-3 **Binärzahlen** (2 Punkte)

Der Algorithmus *Binärzahlen* berechnet die minimale Anzahl der Binärstellen, die zur Darstellung der Zahl z benötigt werden, wobei $z \geq 1$ gilt.

Bei dem im Folgenden benutzten Divisionsoperator *div* wird nur der ganzzahlige Anteil der Division (d.h. ohne Nachkommastellen) als Ergebnis zurückgegeben.

BZ = **function**(binaerzahl:**nat**) **nat**:
  if binaerzahl = 1 **then** 1
  else 1 + BZ(binaerzahl **div** 2)

Andere Lösungsmöglichkeit: mit der Logarithmusfunktion (zur Basis 2), d.h. $\lfloor \log_2(z) \rfloor + 1$ berechnen. Dabei

bezeichnet $\lfloor r \rfloor$ für eine reelle Zahl r den ganzzahligen Anteil von r.

(Eine weitere Alternative ist: $\log_2(z+1)$ berechnen und das Ergebnis aufrunden.)

Bewertung: 1 Punkt für den Anfangsfall, 1 Punkt für die richtige Rekursion. Bei Verwendung der Log-Funktion: 1 Punkte für Berechnung von dem ganzzahligen Anteil von $\log_2(z)$, 1 Punkte für das Addieren von 1. (bzw. 1 Punkte für Berechnung des Logarithmus, 1 Punkte für das Aufrunden.)

Aufgabe 2-4 **Quadratzahltest** (4 Punkte)

Um die gesuchte Funktion definieren zu können, benötigen wir eine allgemeinere Einbettungsfunktion, die für zwei natürliche Zahlen n und k bestimmt, ob n das Quadrat einer Zahl $i \leq k$ ist.

qzt = **function** (n: **nat**, k: **nat**) **bool**:

result true , falls n das Quadrat einer natuerlichen Zahl j $\leq$ k ist, false sonst
  if k = 0 **then** n = k
  else (n = k ∗ k ∨ qzt(n, k − 1))

Mit Hilfe dieser Funktion lässt sich nun die Funktion *istquadrat* wie folgt definieren:

istquadrat = **function** (n: **nat**) **bool**:

result true , falls n Quadratzahl, false andernfalls
  qzt(n , n)

Aufgabe 2-5 **Vollkommenheitstest** (6 Punkte)

Diese Funktion lässt sich nicht direkt durch Rekursion definieren. Wir definieren also zunächst eine Hilfsfunktion, die für zwei natürliche Zahlen $n \geq 1$ und $i < n$ die Summe der Teiler $\leq i$ von n berechnet.

vsumme = **function** (n: **nat**, i: **nat**) **nat**
pre n $\geq$ 1, i < n

result Summe der Teiler $\leq$ i von n
  if i=0 **then** 0
  else if n mod i = 0 **then** vsumme(n, i−1) + i
       else vsumme(n, i−1)

Mit Hilfe der Funktion *vsumme* kann man nun eine Funktion *istvollkommen*, die für eine natürliche Zahl bestimmt, ob sie eine vollkommene Zahl ist, wie folgt definieren:

istvollkommen = **function** (n: **nat**) **bool**:
pre n $\geq$ 1

result true , falls n vollkommen, false sonst
  vsumme(n, n) = 2∗n