Fakultät IV: Elektrotechnik und Informatik
Informatik 1 WS 2001/02,
Jähnichen / Flik / Lechtchinsky / Neunast / Tutoren

Aufgabenblatt 4

letzte Aktualisierung: 15. November, 18:45

Ausgabe: *16.11.2001*
Abgabe: *26./27.11.2001* *Prozent:* *100*

Themen: *Rekursion, Sequenzen von Sequenzen, einfache Algorithmen.*

Der erste von drei Tests findet in den Tutorien am 26./27.11.01 statt.

Hinweise zu den Tests:

Der Tutor entscheidet, ob der Test zu Beginn oder am Ende des Tutoriums geschrieben wird.

Die Bearbeitungszeit beträgt 15 Minuten. Es sind keine Unterlagen zugelassen.
Die Bearbeitung erfolgt einzeln, **nicht** in Gruppen.
Täuschungsversuche werden mit sofortiger Bewertung des Tests als „nicht bestanden“ geahndet.

Grundsätzlich sind für den Erhalt eines Übungsscheins zwei der drei Tests zu bestehen.

Die Themen des ersten Tests orientieren sich am Stoffumfang der Aufgabenblätter 1 bis 4.

Viel Erfolg!

1. Aufgabe (40 Prozent): Sequenzen

1.1. Primfaktorzerlegung (20 Prozent) Deklariert und definiert eine Funktion `primeFactorDecomposition`, die eine natürliche Zahl größer 1 in ihre Primfaktoren zerlegt und diese als Sequenz zurückgibt.

Beispiel: `primeFactorDecomposition`$(314159265) = \langle 3, 3, 5, 7, 127, 7853\rangle$

Hinweis:
Für die Primfaktoren $p_1, \ldots, p_m$ ($m \in \mathbb{N}$ geeignet) von $n \in \mathbb{N}$, $n \geq 2$, gilt:

$$n = \prod_{i=1}^{m} p_i \quad \text{und} \quad p_i = \begin{cases} \min\{k \in \mathbb{N} \mid k \geq 2, k \text{ teilt } n\} & \text{falls } i = 1 \\ \min\{k \in \mathbb{N} \mid k \geq p_{i-1}, k \text{ teilt } \left(n / \prod_{j=1}^{i-1} p_j\right)\} & \text{falls } 1 < i \leq m \end{cases}$$

Ist es erforderlich, in jeder Rekursionsinstanz das ganze Produkt zu bilden? Muß auf der Suche nach dem Minimum jede Zahl geprüft werden oder kann man trivial optimieren?

Lösung:
Erläuterung des Beispiels (das Symbol „$\mid$“ bedeutet „teilt“ und „$\nmid$“ bedeutet „teilt nicht“):
$p_1 = 3$, *da* $2 \nmid 314159265$ *und* $3 \mid 314159265$;
$p_2 = 3$, *da* $314159265/p_1 = 104719755$ *und* $3 \mid 104719755$;

$p_3 = 5$, *da* $104719755/p_2 = 34906585$ *und* $5 \mid 34906585$;
$p_4 = 7$, *da* $34906585/p_3 = 6981317$ *und* $7 \mid 6981317$;
$p_5 = 127$, *da* $6981317/p_4 = 997331$ *und* $\forall k \in \{8, \ldots, 126\} : k \nmid 997331$ *und* $127 \mid 997331$;
$p_6 = 7853$, *da* $997331/p_5 = 7853$ *und* $\forall k \in \{128, \ldots, 7852\} : k \nmid 7853$ *und* $7853 \mid 7853$.

In der Beispielrechnung wurde bereits ausgenutzt, daß das Produkt der Primfaktoren indirekt inkrementell gebildet werden kann, indem die zu zerlegende Zahl durch jeden neu gefundenen Primfaktor dividiert und die Rechnung dann fortgesetzt wird. Außerdem wurde die 6 als potentieller Teiler übersprungen (siehe unten).
Eine direkte Umsetzung des Algorithmus führt auf die Variante:

```
FUN primeFactorDecomposition: nat -> seq[nat]
DEF primeFactorDecomposition(n) == primeFactors(2, n)

-- primeFactors(k, n) berechnet die Sequenz der Primfaktoren
--   von n unter der Voraussetzung, daß n keine Primfaktoren
--   besitzt, die kleiner als k sind.
FUN primeFactors: nat ** nat -> seq[nat]
DEF primeFactors(k, n) ==
  IF   n = 1 THEN <>
  OTHERWISE
  IF n%k = 0 THEN k :: primeFactors(k, n/k)
             ELSE      primeFactors(k+1, n)
  FI
```

Da gerade Zahlen aber keine Primfaktoren darstellen können und die Hälfte der auf die Teilereigenschaft geprüften Zahlen gerade sind, lohnt sich die folgende kleine Optimierung:

```
FUN primeFactorDecomposition: nat -> seq[nat]
DEF primeFactorDecomposition(n) == primeFactors(2, 1, n)

-- primeFactors(k, step, n) berechnet die Sequenz der Primfaktoren
--   von n unter der Voraussetzung, daß n keine Primfaktoren
--   besitzt, die kleiner als k sind.
--   Dabei muß step=1 gelten, wenn k gerade ist
--         und step=2, wenn k ungerade ist.
FUN primeFactors: nat ** nat ** nat -> seq[nat]
DEF primeFactors(k, step, n) ==
  IF   n = 1 THEN <>
  OTHERWISE
  IF n%k = 0 THEN k :: primeFactors(k, step, n/k)
             ELSE      primeFactors(k+step, 2, n)
  FI
```

Anmerkung am Rande: Die Deklaration einer Funktion wie `primeFactors` mit dem ominösen `step`-Parameter gehört in jedem Fall in die Implementation, nicht in die Signatur. Derart seltsame Anwendungsbedingungen darf nur eine interne Hilfsfunktion aufweisen, aber nicht eine Funktion, die als Schnittstelle zur Außenwelt fungiert.

1.2. Aufteilen einer Sequenz (20 Prozent) Deklariert und definiert eine Funktion `ascendingPrefix`, die eine gegebene Sequenz in zwei Teilsequenzen aufteilt, wobei die erste Teilsequenz alle vom Sequenzanfang an aufsteigenden Elemente (monotones Wachstum) und die zweite Teilsequenz den Rest der Sequenz enthält.

Beispiel: `ascendingPrefix`$(\langle 1, 9, 1, 2, 3, 5, 7, 2, 3, 7\rangle) = (\langle 1, 9\rangle, \langle 1, 2, 3, 5, 7, 2, 3, 7\rangle)$

Hinweis: Diese Aufgabe stammt aus der letzten Klausur (Bearbeitungszeit: ca. 12 Minuten).

Lösung:


```
FUN ascendingPrefix: seq[nat] -> seq[nat] ** seq[nat]
DEF ascendingPrefix(S) ==
 IF S <>? THEN (<>,<>)
 ELSE
   LET
     n == ft(S)
     R == rt(S)
   IN
     IF R <>? THEN (S,<>)
     OTHERWISE
     IF n <= ft(R) THEN
       (n::S1, S2)
       WHERE
       (S1, S2) == ascendingPrefix(R)
     ELSE
       (n::(<>), R)
     FI
 FI
```

2. Aufgabe (20 Prozent): Sequenzen von Sequenzen

2.1. Zerlegen einer Sequenz (Tut) Deklariert und definiert eine Funktion `ascendingParts`, die eine gegebene Sequenz in eine Sequenz von Teilsequenzen aufteilt, so daß ihre Verkettung gleich der Ausgangssequenz ist. Dabei sollen die Teilsequenzen jeweils monoton steigend sein.

Beispiel: `ascendingParts`$(\langle 1,9,1,2,3,5,7,2,3,7\rangle) = \langle\langle 1,9\rangle,\langle 1,2,3,5,7\rangle,\langle 2,3,7\rangle\rangle$

Hinweis: Die Funktion `ascendingPrefix` aus Aufgabe 1 könnte hilfreich sein.

Lösung:

```
FUN ascendingParts: seq[nat] -> seq[seq[nat]]
DEF ascendingParts(S) ==
  IF S <>? THEN <>
  ELSE
    LET
      (part, rest) == ascendingPrefix(S)
    IN
      part :: ascendingParts(rest)
  FI
```

2.2. Potenzmenge (Tut) Definiert eine Funktion `FUN powerSet: seq[nat] -> seq[seq[nat]]`, die eine Sequenz auf naive Weise als Repräsentation einer Menge auffaßt und ihre Potenzmenge berechnet (warum ist diese Darstellung des Mengenbegriffs im allgemeinen eine schlechte Wahl?).

Beispiel: $\mathcal{P}(\{1,2,3\}) = \{\emptyset,\{1\},\{2\},\{3\},\{1,2\},\{2,3\},\{1,3\},\{1,2,3\}\}$

Hinweis:

$$\mathcal{P}(M) = \begin{cases} \{\emptyset\} & \text{für } M = \emptyset \\ \mathcal{P}(M \setminus \{x\}) \cup \Big(\bigcup\limits_{P \in \mathcal{P}(M\setminus\{x\})} \{P \cup \{x\}\} \Big) & \text{für } M \neq \emptyset \text{ mit beliebigem } x \in M \end{cases}$$

Lösung:

Sequenzen als Mengen zu „mißbrauchen" ist normalerweise keine gute Idee, da die typischen Mengenoperationen wie $\in$, $\cup$, $\cap$ usw. dann nur aufwendig zu implementieren sind. Insbesondere können zwei gleiche Mengen unterschiedliche Repräsentationen besitzen, denn zwei Sequenzen unterscheiden sich, wenn Elemente permutiert werden, Mengen jedoch nicht. Zumindest müßten die Sequenzen sortiert gehalten werden, um wenigstens die Vergleichsoperation mit begrenztem Aufwand durchführen zu können. Allerdings gewinnt man bei den anderen Operationen dadurch nicht allzuviel, so daß andere Repräsentationen sinnvoller sind (z.B. balancierte Bäume → Info-3).

Erläuterung des Beispiels:

$$\begin{aligned} \mathcal{P}(\{1,2,3\}) &= \mathcal{P}(\{2,3\}) \cup \Big(\bigcup_{P \in \mathcal{P}(\{2,3\})} \{P \cup \{1\}\} \Big) \\ &= \{\emptyset,\{2\},\{3\},\{2,3\}\} \cup \big(\{\emptyset \cup \{1\}\} \cup \{\{2\} \cup \{1\}\} \cup \{\{3\} \cup \{1\}\} \cup \{\{2,3\} \cup \{1\}\}\big) \\ &= \{\emptyset,\{2\},\{3\},\{2,3\},\{1\},\{2,1\},\{3,1\},\{2,3,1\}\} \end{aligned}$$

Eine mögliche Umsetzung der obigen Formel könnte lauten:

```
FUN powerSet: seq[nat] -> seq[seq[nat]]
DEF powerSet ==
  \\S. IF S <>? THEN <> :: <>
       ELSE
         powerSubset ++ prepend(ft(S),powerSubset)
         WHERE
         powerSubset == powerSet(rt(S))
       FI

-- prepend(n,S) stellt jeder in S enthaltenen Sequenz die Zahl n voran
FUN prepend: nat ** seq[seq[nat]] -> seq[seq[nat]]
DEF prepend ==
  \\n,S. IF S <>? THEN <>
                 ELSE (n::ft(S))::prepend(n,rt(S))
         FI
```

Die Verwendung der Listenkonkatenation ++ führt allerdings zu einem verhältnismäßig großen Rechenaufwand (für große Listen), der sich leicht dadurch vermeiden läßt, daß die `prepend`*-Funktion modifiziert wird.*

```
FUN powerSet: seq[nat] -> seq[seq[nat]]
DEF powerSet ==
  \\S. IF S <>? THEN <> :: <>
               ELSE augment(ft(S),powerSet(rt(S)))
       FI

-- augment(n,S) = ( prepend(n,S) vereinigt mit S )
FUN augment: nat ** seq[seq[nat]] -> seq[seq[nat]]
DEF augment ==
  \\n,S. IF S <>? THEN <>
                 ELSE (n::ft(S))::ft(S)::augment(n,rt(S))
         FI
```

2.3. Permutationen einer Sequenz (20 Prozent) Deklariert und definiert eine Funktion `perm`, die alle möglichen Permutationen einer übergebenen Sequenz natürlicher Zahlen generiert. Verwendet **nicht** die Funktion `permutations'SeqOfSeq`.

Beispiel: $\mathbb{P}(\langle 1,2,3\rangle) = \{\langle 1,2,3\rangle, \langle 2,1,3\rangle, \langle 2,3,1\rangle, \langle 1,3,2\rangle, \langle 3,1,2\rangle, \langle 3,2,1\rangle\}$

Hinweis:

$$\mathbb{P}(S) = \begin{cases} \{\langle\rangle\} & \text{für } S = \langle\rangle \\ \bigcup\limits_{\substack{P_1,P_2 \\ P\in\mathbb{P}(S') \\ P=P_1.P_2}} \{P_1.x.P_2\} & \text{für } S = x.S' \text{ mit geeignetem } x \text{ und } S' \end{cases}$$

Lösung:

Erläuterung des Beispiels:

$$\begin{aligned} \mathbb{P}(\langle 1,2,3\rangle) &= \bigcup_{\substack{P_1,P_2 \\ P\in\mathbb{P}(\langle 2,3\rangle) \\ P=P_1.P_2}} \{P_1.1.P_2\} = \bigcup_{\substack{P_1,P_2 \\ P\in\{\langle 2,3\rangle,\langle 3,2\rangle\} \\ P=P_1.P_2}} \{P_1.1.P_2\} \\ &= \{\langle 1,2,3\rangle\} \cup \{\langle 2,1,3\rangle\} \cup \{\langle 2,3,1\rangle\} \cup \{\langle 1,3,2\rangle\} \cup \{\langle 3,1,2\rangle\} \cup \{\langle 3,2,1\rangle\} \\ &= \{\langle 1,2,3\rangle, \langle 2,1,3\rangle, \langle 2,3,1\rangle, \langle 1,3,2\rangle, \langle 3,1,2\rangle, \langle 3,2,1\rangle\} \end{aligned}$$

```
FUN perm: seq[nat] -> seq[seq[nat]]
DEF perm ==
  \\S. IF S <>? THEN <> :: <>
               ELSE insertInAllLists(ft(S),perm(rt(S)))
       FI

-- insertInAllLists(n,S) bildet für jede Sequenz von S alle möglichen
--   Zerlegungen in zwei Teile, fügt jeweils n dazwischen ein und liefert
--   alle resultierenden Sequenzen zurück.
FUN insertInAllLists: nat ** seq[seq[nat]] -> seq[seq[nat]]
DEF insertInAllLists ==
  \\n,S. IF S <>? THEN <>
                 ELSE insertAtAllPositions(n,ft(S))
                      ++ insertInAllLists(n,rt(S))
         FI

-- Für jede mögliche Zerlegung von S = S1.S2 erzeugt
--   insertAtAllPositions(n,S) die Sequenz S' = S1.n.S2 und liefert
--   alle diese Sequenzen zurück.
FUN insertAtAllPositions: nat ** seq[nat] -> seq[seq[nat]]
DEF insertAtAllPositions ==
  \\n,S. IF S <>? THEN (n :: <>) :: <>
                 ELSE (n :: S) :: prepend(ft(S),insertAtAllPositions(n,rt(S)))
         FI
```

3. Aufgabe (40 Prozent): Matrizen als Sequenzen von Sequenzen

Eine Möglichkeit, Matrizen zu repräsentieren, ist es, sie als Sequenzen von Sequenzen darzustellen, z.B. `seq[seq[nat]]`. Wir treffen die Vereinbarung, daß die „inneren" Sequenzen den Zeilen entsprechen und daher gleiche Länge haben müssen. Außerdem machen nur Matrizen mit Dimensionen größer Null einen Sinn.

Beispiel: $\langle\langle 1,2,3,4\rangle, \langle 5,6,7,8\rangle, \langle 9,10,11,12\rangle\rangle$ entspricht $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$

und besitzt in Opal die Darstellung

$(1 :: 2 :: 3 :: 4 :: (\langle\rangle)) :: (5 :: 6 :: 7 :: 8 :: (\langle\rangle)) :: (9 :: 10 :: 11 :: 12 :: (\langle\rangle)) :: (\langle\rangle)$

3.1. Matrix-Vektor-Produkt (Tut) Deklariert und definiert eine Funktion `matVecProd`, die eine Matrix mit einem Vektor multipliziert.

Beispiel: $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 30 \\ 70 \\ 110 \end{pmatrix}$

Hinweis: `skalarProd` von Blatt 3 steht zur Verfügung.

Lösung:

```
-- matVecProd(M,v) setzt voraus, daß die Anzahl der Spalten von M
--   gleich der Anzahl der Elemente (Zeilen) von v ist.
FUN matVecProd : seq[seq[nat]] ** seq[nat] -> seq[nat]
DEF matVecProd(M,v) ==
  IF M <>? THEN <>
          ELSE skalarProd(ft(M),v) :: matVecProd(rt(M),v)
  FI
```

3.2. Matrix-Transposition (20 Prozent) Deklariert und definiert eine Funktion `transp`, die eine Matrix transponiert. Verwendet **nicht** die Funktion `transpose'SeqOfSeq`.

Beispiel: $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}^{\mathrm{T}} = \begin{pmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{pmatrix}$

Lösung:

```
FUN transp: seq[seq[nat]] -> seq[seq[nat]]
DEF transp(M) ==
  IF ft(M) <>? THEN <>
              ELSE mapFt(M) :: transp(mapRt(M))
  FI

-- Extraktion der ersten Spalte einer Matrix
FUN mapFt: seq[seq[nat]] -> seq[nat]
DEF mapFt(M) ==
  IF M <>? THEN <>
          ELSE ft(ft(M)) :: mapFt(rt(M))
  FI

-- Abschneiden der ersten Spalte einer Matrix
--   mit Rückgabe der verbleibenden Untermatrix
FUN mapRt: seq[seq[nat]] -> seq[seq[nat]]
DEF mapRt(M) ==
  IF M <>? THEN <>
          ELSE rt(ft(M)) :: mapRt(rt(M))
  FI
```

3.3. Matrix-Matrix-Produkt (20 Prozent) Deklariert und definiert eine Funktion `matMatProd`, die zwei Matrizen multipliziert.

Beispiel: $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} = \begin{pmatrix} 70 & 80 & 90 \\ 158 & 184 & 210 \\ 246 & 288 & 330 \end{pmatrix}$

Hinweis:
Die mathematische Definition der Ergebnismatrix lautet:

$$C_{m\times n} = A_{m\times l} \cdot B_{l\times n} \quad \text{mit} \quad c_{ij} = \sum_{k=1}^{l} a_{ik} \cdot b_{kj}$$

Nützlicher ist allerdings die Erkenntnis, daß jede Spalte von C durch Bilden des Matrix-Vektor-Produkts von A und der korrespondierenden Spalte von B gebildet wird. Beim Versuch, das umzusetzen, wird man vermutlich auf die Idee kommen, zweimalige Transposition zu Hilfe zu nehmen (warum?). Mit ein bißchen Überlegung genügt aber auch die einmalige Transposition (wie geht das?).

Falls ihr Aufgabe 3.2 nicht lösen konntet, verwendet `transpose'SeqOfSeq`, um diese Aufgabe zu lösen, ansonsten natürlich das eigene `transp`.

Lösung:

```
-- Produkt zweier Matrizen: C = A x B
-- A und B müssen verkettet sein,
--   d.h. Anzahl der Spalten von A = Anzahl der Zeilen von B
FUN matMatProd : seq[seq[nat]] ** seq[seq[nat]] -> seq[seq[nat]]

/*
-- naheliegende Version mit zweimaliger Transposition
DEF matMatProd(M1,M2) ==
  transp(matMatProdHelp(M1,transp(M2)))
*/

-- optimierte Version mit einmaliger Transposition
DEF matMatProd(M1,M2) ==
  matMatProdHelp(transp(M2),M1)

-- Pseudo-Produkt zweier Matrizen:
-- Argumente sind A und B, berechnet wird (A x Bt)t, d.h. B x At
--                                      (t = Transposition)
-- A und Bt müssen verkettet sein
FUN matMatProdHelp : seq[seq[nat]] ** seq[seq[nat]] -> seq[seq[nat]]
DEF matMatProdHelp(M1,M2) ==
  IF M2 <>? THEN <>
            ELSE matVecProd(M1,ft(M2)) :: matMatProdHelp(M1,rt(M2))
  FI
```

Da unsere zeilenorientierte Matrixrepräsentation keinen direkten Zugang zu den Spalten einer Matrix gestattet, berechnet `matMatProdHelp` *nicht direkt das gewünschte Produkt. Vielmehr ist die zweite Matrix als implizit transponiert zu betrachten, da einfach ihre Zeilen so behandelt werden als wären sie Spalten. Analoges gilt für das Ergebnis, das natürlich zeilenweise generiert wird, sich aber eigentlich auf die Spalten bezieht (beim*

Matrizenprodukt bildet die linke Matrix die Spalten*vektoren der rechten Matrix auf die* Spalten*vektoren der Ergebnismatrix ab).*

Mit anderen Worten entspricht `matMatProdHelp` *einer Funktion, die A und B abbildet auf $(A \cdot B^{\mathrm{T}})^{\mathrm{T}}$. Der erste Gedanke zur Kompensation dieser „Unzulänglichkeit" ist wahrscheinlich, die Funktion mit einer transponierten zweiten Matrix aufzurufen und das Ergebnis zu transponieren.*

Besser ist es natürlich, eine kleine mathematische Umformung vorzunehmen, denn allgemein gilt $(A \cdot B)^{\mathrm{T}} = B^{\mathrm{T}} \cdot A^{\mathrm{T}}$ und $(A^{\mathrm{T}})^{\mathrm{T}} = A$:

$$(A \cdot B^{\mathrm{T}})^{\mathrm{T}} = (B^{\mathrm{T}})^{\mathrm{T}} \cdot A^{\mathrm{T}} = B \cdot A^{\mathrm{T}}$$

Also kann man problemlos mit einer einzigen Transposition auskommen.

Musterlösung