Eine Sache lernt man, indem man sie macht.
Cesare Pavese (1908-1950)
italien. Schriftsteller

Für das Können gibt es nur einen Beweis, das Tun.
Marie von Ebner-Eschenbach (1830-1916)
österr. Schriftstellerin

6. Aufgabenblatt zu Funktionale Programmierung von Mo, 18.11.2019.
Erstabgabe: Mo, 25.11.2019 (12:00 Uhr)

Themen: *Funktionen höherer Ordnung, Rechnen mit Funktionen, Funktionen als Argument und Resultat*

Besprechung von Aufgabenlösungen, Zweitabgabefrist

- **Teil A, programmiertechnische Aufgaben:** Am ersten Übungsgruppentermin, der auf die *Zweitabgabe* der programmiertechnischen Aufgaben folgt.
- Teil B, Papier- und Bleistiftaufgaben: Entfällt auf den Angaben 5 bis 7.
- **Teil C, Terminhinweise:** Für Übungsgruppen, Vorlesung, Tutorensprechstunde.
- **Zweitabgabefrist:** Siehe „Hinweise zu Organisation und Ablauf der Übung“ auf der Webseite der Lehrveranstaltung.

A Programmiertechnische Aufgaben (beurteilt, max. 100 Punkte)

Schreiben Sie zur Lösung der folgenden Aufgaben ein gewöhnliches Haskell-Skript und legen Sie es in einer (Abgabe-) Datei namens `Angabe6.hs` in Ihrem Home-Verzeichnis auf der Maschine `g0` ab. Der Name der Abgabedatei ist für Erst- und Zweitabgabe ident.

Kommentieren Sie Ihr Programm wieder zweckmäßig, aussagekräftig und problemangemessen. Benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten. Versehen Sie alle Funktionen, die Sie zur Lösung der Aufgaben benötigen, mit ihren Typdeklarationen, d.h. geben Sie deren syntaktische Signatur oder kurz, Signatur, explizit an.

A.1 Wir führen eine einfache imperative Programmiersprache ein, für die wir uns die auf Angabe 5 eingeführten arithmetischen und logischen Ausdrücke sowie die darauf implementierten Auswertungsfunktionen zunutze machen. In der Folge wiederholen wir diese Deklarationen und erweitern sie um die unserer Programmiersprache. Nehmen Sie bei Bedarf weitere Instanzbildungen für Typklassen vor:

```
data Arith_Variable = A1 | A2 | A3 | A4 | A5 | A6 deriving (Eq,Show)
data Log_Variable   = L1 | L2 | L3 | L4 | L5 | L6 deriving (Eq,Show)

data Arith_Ausdruck = AK Int                -- Arithmetische Konstante
                    | AV Arith_Variable  -- Arithmetische Variable
                    | Plus Arith_Ausdruck Arith_Ausdruck  -- Addition
                    | Minus Arith_Ausdruck Arith_Ausdruck -- Subtraktion
                    | Mal Arith_Ausdruck Arith_Ausdruck   -- Multiplikation
                    deriving (Eq,Show)

data Log_Ausdruck = LK Bool                              -- Logische Konstante
                  | LV Log_Variable                      -- Logische Variable
```

```
                      | Nicht Log_Ausdruck              -- Logische Negation
                      | Und Log_Ausdruck Log_Ausdruck   -- Logische Konjunktion
                      | Oder Log_Ausdruck Log_Ausdruck  -- Logische Disjunktion
                      | Gleich Arith_Ausdruck Arith_Ausdruck  -- Wertgleichheit
                                                              -- arith. Ausdruecke
                      | Kleiner Arith_Ausdruck Arith_Ausdruck -- Linker Ausdruck
                                                              -- echt wertkleiner
                                                              -- als rechter
                                                              -- Ausdruck
                      deriving (Eq,Show)

type Arith_Variablenbelegung = Arith_Variable -> Int  -- Total definierte Abb.
type Log_Variablenbelegung   = Log_Variable -> Bool   -- Total definierte Abb.
type Variablenbelegung       = (Arith_Variablenbelegung,Log_Variablenbelegung)

links :: (Either a b) -> a
links (Left x) = x

rechts :: (Either a b) -> b
rechts (Right y) = y

class Evaluierbar a where
 evaluiere :: a -> Variablenbelegung -> Either Int Bool

type Adresse = Int
type Sprungadresse = Adresse

data Anweisung = AZ Arith_Variable Arith_Ausdruck -- Wertzuweisung an
                                                  -- arithmetische Variable
                 | LZ Log_Variable Log_Ausdruck   -- Wertzuweisung an
                                                  -- logische Variable
                 | FU Log_Ausdruck Sprungadresse Sprungadresse -- Fallunter-
                                                               -- scheidung
                 | BS Log_Ausdruck Sprungadresse  -- Bedingter Sprung
                 | US Sprungadresse               -- Unbedingter Sprung
                 | MP Adresse Anweisung           -- Selbstmodifikation des
                                                  -- Programms

type Zustand         = Variablenbelegung
type Anfangszustand  = Zustand
type Endzustand      = Zustand
type Zwischenzustand = Zustand
type Programm        = [Anweisung]
type EPS             = Programm
```

Schreiben Sie zwei Interpretierervarianten für Programme π der *Einfachen Programmiersprache* EPS:

```
interpretiere_1 :: EPS -> Anfangszustand -> Endzustand
interpretiere_2 :: EPS -> Anfangszustand -> [Zwischenzustand]
```

Angesetzt auf ein EPS-Programm π und einen Anfangszustand σ liefert

(a) `interpretiere_1` den Endzustand σ', in dem die Ausführung von π durch den Interpretierer endet, wenn diese terminiert. Terminiert die Ausführung nicht, ist das Verhalten von `interpretiere_1` undefiniert; ein besonderes Verhalten oder Resultat ist in diesem Fall von `interpretiere_1` nicht gefordert.

(b) `interpretiere_2` die Liste der durch Wertzuweisungen an arithmetische oder logische Variablen (möglicherweise) geänderten (Zwischen-) Zustände (die Ausführung von Fallunterscheidungen, Sprüngen und Modifikationsanweisungen ändert einen Zustand nicht, s.u.). Der erste Zustand in dieser Liste (d.h. an Indexposition 0) ist der Anfangszustand σ. Die von `interpretiere_2` gelieferte (Zwischen-) Zustandsliste ist endlich, wenn die Ausführung von π angesetzt auf σ terminiert, sonst (konzeptuell) unendlich (in diesem Fall terminiert `interpretiere_2` nicht und bricht schließlich irregulär ab, wenn der gesamte Maschinenspeicher verbraucht ist).

Die Arbeitsweise der beiden Interpretierervarianten und die Bedeutung der Anweisungen eines EPS-Programms sind dabei wie folgt festgelegt, wobei arithmetische und logische Ausdrücke mithilfe von `evaluiere` ausgewertet werden:

- Beide Interpretierervarianten beginnen die Ausführung eines EPS-Programms π mit der Anweisung, die an Indexposition 0 von π steht.
- Ist die Anweisung, die der Interpretierer aktuell ausführt, ein(e)
  - *Wertzuweisung* an eine arithmetische oder logische Variable, so wertet der Interpretierer den (rechtsseitigen) Ausdruck in der aktuellen Variablenbelegung aus und weist der (linksseitigen) Variable den entsprechenden Wert zu, wodurch eine neue Variablenbelegung entsteht. Anschließend führt der Interpretierer die Ausführung von π mit der Anweisung an der um 1 höheren Indexposition von π unter der neuen Variablenbelegung fort.
  - *Fallunterscheidung*, so wertet der Interpretierer den logischen Ausdruck in der aktuellen Variablenbelegung aus. Liefert diese Auswertung den Wert `True`, so führt der Interpretierer die Ausführung von π mit der Anweisung an der durch den Wert der linken Sprungadresse in der aktuellen Variablenbelegung bezeichneten Indexposition von π mit unveränderter Variablenbelegung fort. Liefert die Auswertung den Wert `False`, so führt der Interpretierer die Ausführung von π mit der Anweisung an der durch den Wert der rechten Sprungadresse in der aktuellen Variablenbelegung bezeichneten Indexposition von π mit unveränderter Variablenbelegung fort.
  - *bedingter Sprung*, so wertet der Interpretierer den logischen Ausdruck in der aktuellen Variablenbelegung aus. Liefert diese Auswertung den Wert `True`, so führt der Interpretierer die Ausführung von π mit der Anweisung an der durch den Wert der Sprungadresse in der aktuellen Variablenbelegung bezeichneten Indexposition von π fort. Liefert die Auswertung den Wert `False`, so führt der Interpretierer die Ausführung von π mit der Anweisung an der um 1 höheren Indexposition von π mit unveränderter Variablenbelegung fort.
  - *unbedingter Sprung*, so führt der Interpretierer die Ausführung von π mit der Anweisung an der durch den Wert der Sprungadresse in der aktuellen Variablenbelegung bezeichneten Indexposition von π mit unveränderter Variablenbelegung fort.

– *Modifikationsanweisung*, so wertet der Interpretierer den arithmetischen Ausdruck der Modifikationsanweisung zum Wert w aus. Erfüllt w die Ungleichung $0 \leq w \leq \texttt{length}(\pi) - 1$, so ersetzt der Interpretierer die an Indexposition w von π stehende Anweisung durch die in der Modifikationsanweisung gegebene Anweisung. Ist $w = \texttt{length}(\pi)$, so verlängert der Interpretierer π um eine Anweisung, in dem die in der Modifikationsanweisung gegebene Anweisung als (neue) letzte Anweisung von π angehängt wird. Anschließend setzt der Interpretierer die Ausführung des Programms mit der an der Indexposition w stehenden Anweisung fort. Erfüllt w die Ungleichung $0 \leq w \leq \texttt{length}(\pi)$ nicht, so hat die Modifikationsanweisung keinen Effekt und der Interpretierer setzt die Programmausführung von π mit der Anweisung an der um 1 höheren Indexposition fort.

Die Ausführung von π durch den Interpretierer endet, wenn die Position des auszuführenden Befehls keine gültige Indexposition von π ist.

Bei der Implementierung dürfen Sie davon ausgehen, dass Belegungen für arithmetische und logische Variablen und Zustände stets total definiert sind. Implementieren Sie auch die Funktionen `fib` und `fac` wie in den folgenden Aufrufbeispielen angegeben.

Aufrufbeispiele:

```
fib n
 | n == 0 = 0
 | n == 1 = 1
 | n > 0  = fib (n-2) + fib (n-1)
 | True   = n

fac n
 | n == 0 = 1
 | n > 0  = n * fac (n-1)
 | True   = n

-- Programm pi0: Berechnung der Fakultaet fuer den Anfangswert von A1, falls
-- dieser groesser oder gleich 0 ist, in A2:
pi0 = [AZ A2 (AK 1),BS (Gleich (AV A1) (AK 1)) 999,
       AZ A2 (Mal (AV A2) (AV A1)),AZ A1 (Minus (AV A1) (AK 1)),US 1]

lvb0 = \ lv -> True :: Log_Variablenbelegung
avb0 = \ av -> if av == A1 then fib 4 else fib 6 - (fac 3 + 2)
       :: Arith_Variablenbelegung
zst0 = (avb0,lvb0) :: Anfangszustand

avb1 = \ av -> if av == A2 then 1 else avb0 av
avb2 = \ av -> if av == A2 then 3 else avb0 av
avb3 = \ av -> case av of A1 -> 2
                          A2 -> 3
                          x  -> avb0 x
avb4 = \ av -> case av of A1 -> 2
                          A2 -> 6
                          x  -> avb0 x
```

```
avb5 = \ av -> case av of A1 -> 1
                          A2 -> 6
                          x  -> avb0 x

interpretiere_1 pi0 zst0 ->> (avb5,lvb0) -- Informell, keine direkte Ausgabe
                                         --  am Bildschirm moeglich (s.a. A.2).
interpretiere_2 pi0 zst0 ->> [(avb0,lvb0),(avb1,lvb0),(avb2,lvb0),
                              (avb3,lvb0),(avb4,lvb0),(avb5,lvb0)] -- s.o.
take 2 (drop 3 (interpretiere_2 pi0 zst0)) ->> [(avb3,lvb0),(avb4,lvb0)] -- s.o.

-- Unmittelbare Bildschirmausgaben moeglich z.B. fuer:
fst (interpretiere_1 pi0 zst0) A1 ->> fst (avb5,lv0) A1 ->> avb5 A1 ->> 1
fst (interpretiere_1 pi0 zst0) A2 ->> fst (avb5,lv0) A2 ->> avb5 A2 ->> 6
fst (interpretiere_1 pi0 zst0) A3 ->> fst (avb5,lv0) A3 ->> avb5 A3 ->> 0
fst (interpretiere_1 pi0 zst0) A4 ->> fst (avb5,lv0) A4 ->> avb5 A4 ->> 0
fst (interpretiere_1 pi0 zst0) A5 ->> fst (avb5,lv0) A5 ->> avb5 A5 ->> 0
fst (interpretiere_1 pi0 zst0) A6 ->> fst (avb5,lv0) A6 ->> avb5 A6 ->> 0

fst ((interpretiere_2 pi0 zst0) !! 0) A1 ->> ... ->> avb0 A1 ->> 3
fst ((interpretiere_2 pi0 zst0) !! 1) A6 ->> ... ->> avb1 A6 ->> 0
fst ((interpretiere_2 pi0 zst0) !! 2) A1 ->> ... ->> avb2 A1 ->> 3
fst ((interpretiere_2 pi0 zst0) !! 3) A2 ->> ... ->> avb3 A2 ->> 3
fst ((interpretiere_2 pi0 zst0) !! 4) A1 ->> ... ->> avb4 A1 ->> 2
fst ((interpretiere_2 pi0 zst0) !! 5) A1 ->> ... ->> avb5 A1 ->> 1
fst ((interpretiere_2 pi0 zst0) !! 5) A2 ->> ... ->> avb5 A2 ->> 6
fst ((interpretiere_2 pi0 zst0) !! 5) A3 ->> ... ->> avb5 A3 ->> 0

-- Programm pi1: Leeres Programm
pi1 = []
interpretiere_1 pi1 zst0 ->> zst0   -- s.o.
interpretiere_2 pi1 zst0 ->> [zst0] -- s.o.

-- Programm pi2: Selbstmodifizierendes Programm
pi2 = [MP ((fib (fac 3)) - (fac (1+2) + 2)) (AZ A3 (AK (5*(fib (fac 3))+2)))]

avb6 = \ av -> 1 :: Arith_Variablenbelegung
zst1 = (avb6,lvb0) :: Anfangszustand
zst2 = (\ av -> if av == A3 then (fac 3) * (fib 7 - fac 3) else avb6 av,
        \ lv -> if lv /= L6
                  then (True /= False) && True
                  else (mod (4 * (avb6 A3)) (2 + fac (avb6 A1)) /= 0) )
       :: Anfangszustand
interpretiere_1 pi2 zst1
 ->> (\ av -> if av == A3 then 42 else fst zst1 av,snd zst1)        -- s.o.
interpretiere_2 pi2 zst1
 ->> [zst1,(\ av -> if av == A3 then 42 else fst zst1 av,snd zst1)] -- s.o.
interpretiere_1 pi2 zst2 `==` interpretiere_1 pi2 zst1 -- symbolisch: (Listen
tail (interpretiere_2 pi2 zst2)                        -- von) Fkt. nicht in Eq
 `==` tail (interpretiere_2 pi2 zst1)
```

```
-- Programm pi3: Selbstmodifizierendes, nichtterminierendes Programm
pi3 = [MP ((fac 3 + 2) - (fib (fac 3)))
          (BS (LK ((fac 5) > (fib 5))) (fib (fac 3) - (fac 3 + 2)))]
      ++ pi0 ++ pi2

interpretiere_1 pi3 zst1 ->> ‘undefiniert wg. Nichtterminierung’
interpretiere_2 pi3 zst1 ->> [zst1.. -- terminiert nicht, aber fuehrt keine
                                     -- zustandsveraendernde Anweisung aus;
                                     -- deshalb erscheint nur das erste Listen-
                                     -- element gefolgt von unendlichem Warten.
drop 2 (interpretiere_2 pi3 zst1) ->> ...
   -- unendliches Warten, da interpretiere_2 nicht terminiert,
   -- aber nach dem ersten keine weiteren Listenelemente liefert.

-- Programm pi4: Nichtterminierendes, kontinuierlich zustands-
-- veraenderndes Programm
pi4 = [LZ L1 (Nicht (LV L1)), BS (LK True) 0]
interpretiere_1 pi4 zst1 ->> ‘undefiniert wg. Nichtterminierung’
interpretiere_2 pi4 zst1 ->> [zst1,zst2,zst1,zst2,zst1,zst2,..
    -- terminiert nicht regulaer, sondern irregulaer, wenn der gesamte
    -- Maschinenspeicher aufgebraucht ist
 wobei zst2 = (fst zst1,\ lv -> if lv == L1 then False else True)

take 4 (interpretiere_2 pi4 zst1)
 ->> [zst1,zst2,zst1,zst2] -- nicht unmittelbar ausgebbar, s.o.
fst (take 4 (interpretiere_2 pi4 zst1) !! 3) A3 ->> ... ->> 1
snd (take 4 (interpretiere_2 pi4 zst1) !! 1) L1 ->> ... ->> False
snd (take 4 (interpretiere_2 pi4 zst1) !! 1) L2 ->> ... ->> True

-- Programm pi5: Nichtterminierendes, formal (nicht tatsaechlich) zustands-
-- veraenderndes Programm (der Zustand wird ident ueberschrieben)
pi5 = [LZ L1 (LV L1), BS (LK True) 0]
interpretiere_1 pi5 zst1 ->> ‘undefiniert wg. Nichtterminierung’
interpretiere_2 pi5 zst1 ->> [zst1,zst1,zst1,zst1,zst1,..
    -- terminiert nicht regulaer, sondern irregulaer, wenn der gesamte
    -- Maschinenspeicher aufgebraucht ist
 wobei zst2 = (fst zst1,\ lv -> if lv == L1 then False else True)

take 5 (interpretiere_2 pi5 zst1)
 ->> [zst1,zst1,zst1,zst1,zst1] -- nicht unmittelbar ausgebbar, s.o.
fst (take 5 (interpretiere_2 pi5 zst1) !! 3) A3 ->> ... ->> 1
snd (take 4 (interpretiere_2 pi5 zst1) !! 1) L1 ->> ... ->> True
snd (take 4 (interpretiere_2 pi5 zst1) !! 1) L2 ->> ... ->> True
snd (take 5 (interpretiere_2 pi5 zst1) !! 2) L5 ->> ... ->> True
```

A.2 Da unsere Sprachen für arithmetische und logische Ausdrücke jeweils nur über endlich vielen Variablen aufgebaut sind, können wir Variablenbelegungen auch in Form aufsteigend nach Namensindizes geordneter Listen von Variable/Wert-Paaren ausgeben. Schreiben Sie entsprechende Funktionen, die das leisten:

```
gib_aus_arith_Varbel :: Arith_Variablenbelegung -> [(Arith_Variable,Int)]
gib_aus_log_Varbel :: Log_Variablenbelegung -> [(Log_Variable,Bool)]
gib_aus_Zustand :: Zustand -> ([(Arith_Variable,Int)],[(Log_Variable,Bool)])
```

Aufrufbeispiele:

```
avb = \ av -> 42   :: Arith_Variablenbelegung
lvb = \ lv -> True :: Log_Variablenbelegung
zst = (avb,lvb)    :: Zustand

gib_aus_arith_Varbel avb
 ->> [(A1,42),(A2,42),(A3,42),(A4,42),(A5,42),(A6,42)]

gib_aus_log_Varbel lvb
 ->> [(L1,True),(L2,True),(L3,True),(L4,True),(L5,True),(L6,True)]

gib_aus_Zustand zst
 ->> ([(A1,42),(A2,42),(A3,42),(A4,42),(A5,42),(A6,42)],
      [(L1,True),(L2,True),(L3,True),(L4,True),(L5,True),(L6,True)])

gib_aus_Zustand (interpretiere_1 pi2 zst1)
 ->> ([(A1,1),(A2,1)(A3,42)(A4,1)(A5,1)(A6,1)],
      [(L1,True),(L2,True),(L3,True),(L4,True),(L5,True),(L6,True)])
```

A.3 Schreiben Sie zwei `EPS`-Programme, definieren Sie Variablenbelegungen und implementieren Sie eine Funktion zur Erzeugung von Zuständen, um die Funktionen aus A.1 und A.2 umfassender testen zu können. Im einzelnen:

(a) Schreiben Sie `EPS`-Programme für folgende Aufgaben:

i. `ggt` berechnet angewendet auf zwei natürliche Zahlen $m, n \in IN_1$ den größten gemeinsamen Teiler von m und n. Die Werte für m und n werden dabei im Anfangszustand mit der Belegung der Variablen `A1` und `A2` übergeben. Bei Terminierung von `ggt` soll die Variable `A3` im Endzustand mit dem Wert des größten gemeinsamen Teilers von m und n belegt sein. Für die Belegungen aller anderen (logischen und arithmetischen) Variablen ist nichts gefordert. Sind `A1` und `A2` im Anfangszustand nicht mit Werten größer oder gleich 1 belegt, ist kein bestimmtes Verhalten von `ggt` gefordert.

```
ggt :: EPS
ggt = ...
```

ii. `fibo` berechnet angewendet auf eine ganze Zahl n den Wert der Fibonacci-Funktion für n, falls $n \geq 0$ ist. Ist $n < 0$ liefert `fibo` den Wert der Fibonacci-Funktion für $-n$. Zusätzlich zeigt `fibo` an, dass der Aufruf für die Ausführung modifiziert worden ist. Der Wert von n wird dabei im Anfangszustand mit der Belegung der Variablen `A1` übergeben. Ist dieser Wert größer oder gleich 0, terminiert `fibo` und für den Endzustand gilt: Die Variable `A6` ist mit dem Wert der Fibonacci-Funktion für n belegt und die Variable `L1` mit dem Wert `True`. War der Anfangswert von `A1` kleiner als 0, so terminiert `fibo` ebenfalls und für den Endzustand gilt: Die Variable `A6` ist mit dem Wert der Fibonacci-Funktion für $-n$ belegt und die Variable `L1` mit dem Wert `False`.

```
fibo :: EPS
fibo = ...
```

(b) Vervollständigen Sie die Zustandsvereinbarungen von `azst1` und `azst2`:

i. Die arithmetischen Variablen `A1` und `A2` sollen in `azst1` mit den Werten 24 und 60 belegt sein. Alle anderen arithmetischen Variablen mit dem Wert 0, alle logischen Variablen mit dem Wert `True`.

```
azst1 :: Anfangszustand
azst1 = ...
```

ii. Die arithmetischen Variablen `A1` und `A2` sollen in `azst2` mit den Werten 18 und 45 belegt sein. Alle anderen arithmetischen Variablen mit dem Wert ihres "Namensindex", `A3` also mit dem Wert 3, `A4` mit dem Wert 4, usw., alle logischen Variablen mit geradem Namensindex (`L2`, `L4`, `L6`) mit dem Wert `True`, mit ungeradem Namensindex (`L1`, `L3`, `L5`) mit dem Wert `False`.

```
azst2 :: Anfangszustand
azst2 = ...
```

(c) Schreiben Sie eine Funktion `generiere`, die Zustände aus Wertelisten erzeugt:

```
generiere :: [Int] -> [Bool] -> Zustand
```

Angewendet auf eine Liste `ns` ganzer Zahlen und `bs` Wahrheitswerte, erzeugt `generiere` einen Zustand, in dem die arithmetischen und logischen Variablen aufsteigend nach Namensindizes mit den Werten der ersten 6 Elemente von `ns` bzw. `bs` belegt sind. Überschüssige Elemente in `ns` und `bs` werden ignoriert. Enthalten die Listen `ns` oder `bs` weniger als 6 Elemente, so werden überschüssige arithmetische Variablen mit dem Wert 0 und überschüssige logische Variablen mit dem Wert `False` belegt.

Testen Sie alle Funktionen auch mit weiteren eigenen `EPS`-Programmen (z.B. Multiplikation zweier ganzer Zahlen, Potenzfunktion, Binomialkoeffizienten,...) und Anfangszuständen; schreiben Sie auch Testprogramme, die die Belegung der logischen Variablen abfragen und ändern (z.B. exklusives oder, *nand*-Funktion, *nor*-Funktion,...) (ohne Abgabe).

Wichtig:

1. **Wiederverwendung früherer Lösungsteile:** Wenn Sie einzelne Rechenvorschriften aus früheren Lösungen wieder verwenden möchten, kopieren Sie diese bitte in die neue Abgabedatei ein. Ein `import` (eigener Module) schlägt für die Auswertung durch das Abgabeskript fehl, weil Ihre alte Lösung, aus der importiert wird, nicht mit abgesammelt wird. Deshalb: Kopieren statt importieren zur Wiederwendung!

2. **Abgaben auf `g0` auf gewünschtes Verhalten prüfen:** Aufgabenlösungen werden stets auf der Maschine `g0` unter Hugs überprüft. Überzeugen Sie sich deshalb bitte, dass Ihre Lösungen für dieses und auch alle weiteren Aufgabenblätter sich auf der `g0` unter Hugs wie von Ihnen erwartet und gewünscht verhalten; überzeugen Sie sich bei jeder Abgabe davon. Das gilt besonders, wenn Sie für die Entwicklung Ihrer Haskell-Programme mit einem anderen Werkzeug oder einer anderen Maschine als der `g0` arbeiten!

B Papier- und Bleistiftaufgaben

Entfällt auf den Angaben 5 bis 7.

C Terminhinweise

1. **Übungsgruppen**: In dieser Woche finden keine Übungsgruppentreffen statt. Die nächsten Übungsgruppentreffen sind in der kommenden Woche von Montag, 25.11.2019, bis Donnerstag, 28.11.2019.
2. **Vorlesung:** Der nächste Vorlesungstermin ist ebenfalls in der kommenden Woche, am Dienstag, den 26.11.2019, von 08:15 Uhr bis 09:45 Uhr im Informatik-Hörsaal in der Treitlstraße.
3. **Tutorensprechstunde:** Die nächste Tutorensprechstunde ist am Freitag dieser Woche, den 22.11.2019, im `complang`-Labor. Sie finden das `complang`-Labor im Innenhof des Institutsgebäudes in der Argentinierstraße 8.

Alle weiteren geplanten Vorlesungs- und Übungsgruppentermine finden Sie auf der Webseite zur Lehrveranstaltung. Mögliche Änderungen werden dort oder/und in der Vorlesung bzw. in den von möglichen Änderungen betroffenen Übungsgruppen bekanntgegeben.