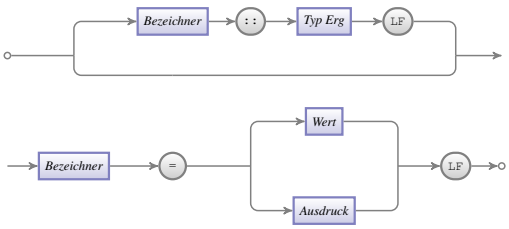
Hinweise

i). Kommentare

- Zeilenkommentar: `--Text`
- Blockkommentar: `{- Text -}`

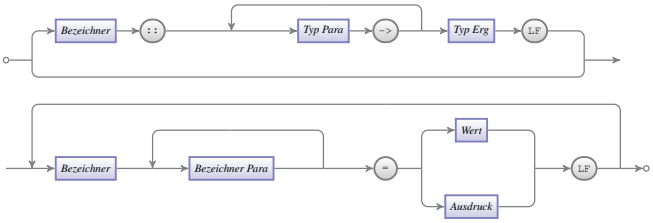
ii). Konstanten

```
pi :: Double
pi = 3.14
```



iii). Funktionen

```
plus :: Int -> Int -> Int
plus a b = a + b
```

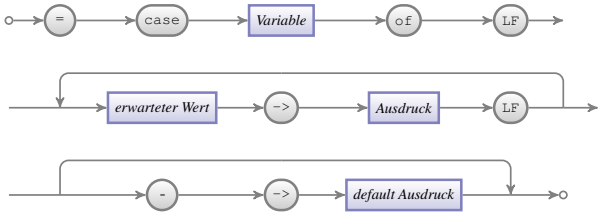


iv). If-Then-Else

```
ggT a b = if b == 0
          then a
          else ggT b (mod a b)
```

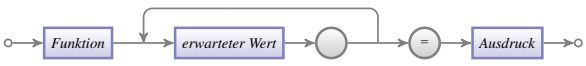
v). Case Of

```
ggT a b = case b of
          0 -> a
          _ -> ggT b (mod a b)
```



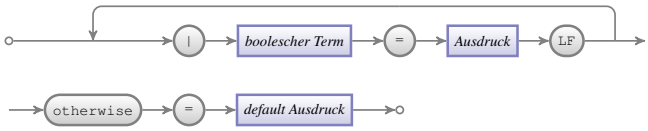
vi). Pattern Matching

```
ggT a 0 = a
ggT a b = ggT b (mod a b)
```



vii). Guards

```
ggT a b | b == 0     = a
        | otherwise =
            ggT b (mod a b)
```



viii). Module

```
module Wurf(weite, square) where
weite :: Double -> Double -> Double
weite v0 phi = ((square v0) / 9.81) * sin (2 * phi)
square :: Double -> Double
square x = x * x

module Foo where
import Wurf hiding (weite)
bar ... = ... (square a) ...
```

ix). Eigene Datentypen

```
data Point = Point{x :: Double, y :: Double}

data Shape = Circle{center :: Point,
    radius :: Double}
  | Rectangle{point :: Point, width :: Double,
    height :: Double}
  | Triangle{point1 :: Point, point2 :: Point,
    point3 :: Point}
```

x). Typparameter

Keine Typparameter in Datentyp Definitionen!

```
data (Eq a, Ord a) => Pair a b =
  PairConst {first :: a, second :: b} deriving(Show)
```

besser:

```
data Pair a b = PairConst a b deriving(Show)

instance Eq a => Eq (Pair a b) where
  (PairConst i1 _) == (PairConst i2 _) = i1 == i2

instance Ord a => Ord (Pair a b) where
   (PairConst i1 _) <= (PairConst i2 _) = i1 <= i2

bubbleSort :: (Ord a) => [(Pair a b)] -> [(Pair a b)]
bubbleSort [] = []
bubbleSort (x:xs) = step $ foldl go (x,[]) xs where
  go (y,acc) x = (min x y, max x y : acc)
  step (x,acc) = x : bubbleSort acc
```

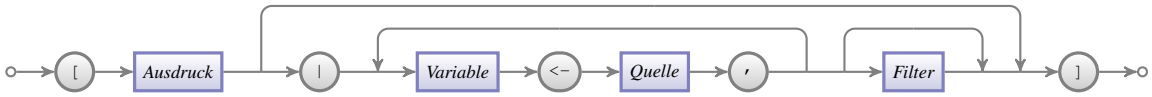
$ ist nichts anderes als:

```
(7 + 6) / (5 + 3) = (7 + 6) / $ 5 + 3
```

Die „Klammerung“ geht bis zum Zeilenende.

xi). Listen

```
primes = sieve [2..]
  where
    sieves (p:xs) = p:sieves [x|x<- xs, mod x p > 0]
```



xii). Operationen

Bezeichner	Typ	Bedeutung
(+)	a −> a −> a	Addition
(−)	a −> a −> a	Subtraktion
(∗)	a −> a −> a	Multiplikation
negate	a −> a	Negation
abs	a −> a	Absolutbetrag
signum	a −> a	Vorzeichenbildung

Tabelle 1: Für die Typen Int, Integer, Float und Double

Bezeichner	Typ	Bedeutung
succ	a −> a	Nachfolgerbildung
pred	a −> a	Vorgängerbildung
div	a −> a −> a	ganzzahlige Division, Ergebnis wird abgerundet
mod	a −> a −> a	zur ganzzahligen Division div gehörender Rest
quot	a −> a −> a	ganzzahlige Division, Ergebnis wird Richtung 0 gerundet
rem	a −> a −> a	zur ganzzahligen Division quot gehörender Rest

Tabelle 2: Für die Typen Int und Integer

Bezeichner	Typ	Bedeutung
(/)	a −> a −> a	Division
recip	a −> a	Kehrwertbildung
(∗∗)	a −> a −> a	Potenzieren
sqrt	a −> a	Ziehen der Quadratwurzel

Tabelle 3: Für die Typen Float und Double

Bezeichner	Typ	Bedeutung
(||)	a −> a −> a	Oder
(&&)	a −> a −> a	Und
not	a −> a	Verneinung

Tabelle 4: Für den Typ Bool

1. Einfache Haskell-Funktion

Im Folgenden soll eine Funktion programmiert werden, die die Wegstrecke $s(t)$ berechnet, welche im freien Fall unter Einfluss der Erdbeschleunigung $g = 9.81$ nach einer Zeit t zurück gelegt wurde. Verwenden Sie die Formel

$$s(t) = \frac{1}{2} \cdot gt^2.$$

Die Wegstrecke soll in Metern und die Zeit in Sekunden angegeben werden.

a).

Mit welche Datentypen sollten die physikalischen Größen Weg, Zeit und Beschleunigung repräsentiert werden? Geben Sie den daraus resultierenden Typ einer Funktion `fallstrecke` an, die $s(t)$ aus t berechnet.

b).

Schreiben Sie die Typsignatur `fallstrecke :: Typ` sowie die Definition von `fallstrecke` in eine Datei. Laden Sie diese in GHCi und testen Sie Ihre Funktion. Zum Beispiel sollte sich `fallstrecke 1` = 4.905 ergeben.

c).

Verändern Sie Ihr Programm zunächst so, dass g als globale Variable (Konstante) deklariert ist. Ersetzen Sie dann die globale Deklaration durch lokale Deklarationen, zunächst mit `where` und anschließend mit `let`.

d).

Die Eingabe eines negativen Wertes soll zu einem undefinierten Funktionswert und zu einer Fehlermeldung führen. Dies erreichen Sie mit Hilfe der Funktion `error`, z.B. durch `error "negative Zeit"`. Realisieren Sie dieses Verhalten einmal mit **If-Then-Else Ausdrücken** und einmal mit **Guards**.

2. Einfache Haskell-Funktionen

Schreiben Sie Haskell-Funktionen zur Berechnung von Volumen, Oberflächeninhalt und Radius von Kugeln. Beachten Sie, dass einige dieser Funktionen als Komposition der übrigen aufgefasst werden können. Verwenden Sie die vordefinierte Konstante `pi :: Double`. Hilfreiche Funktionen über reellen Zahlen finden Sie in den Hinweisen dieses Aufgabenblattes.

a).

Programmieren Sie die folgenden Funktionen:

- `radiusNachOberflaeche :: Double -> Double`
- `oberflaecheNachRadius :: Double -> Double`
- `radiusNachVolumen :: Double -> Double`
- `volumenNachRadius :: Double -> Double`
- `oberflaecheNachVolumen :: Double -> Double`
- `volumenNachOberflaeche :: Double -> Double`

b).

Da es keine Kugeln mit negativem Volumen, Oberflächeninhalt oder Radius gibt, sollen die Funktionen für negative Eingaben undefiniert bleiben und mit einer Fehlermeldung abbrechen. Erweitern Sie dazu die Funktionen um sinnvolle Fallunterscheidungen und Fehlermeldungen.

3. Bbedingte Ausdrücke

Schreiben Sie eine Funktion vom Typ Integer −> Bool, die zu einer gegebenen Jahreszahl prüft, ob sie im Gregorianischen Kalender ein Schaltjahr ist oder nicht.

Pseudocode:

```
if year is >= 1582 then
  if year is divisible by 400 then
    is_leap_year
  else if year is divisible by 100 then
    not_leap_year
  else if year is divisible by 4 then
    is_leap_year
  else
    not_leap_year
else
  not_leap_year
```

a).

Verwenden Sie hierfür nur If-Then-Else-Ausdrücke.

b).

Verwenden Sie hierfür nur Guards

c).

Verwenden Sie hierfür weder If-Then-Else-Ausdrücke noch Guards, sondern die booleschen Operatoren.

4. Lokale Deklarationen

a).

Zur Bestimmung der Nullstellen von quadratischen Gleichungen der Form $ax^2 + bx + c$ gibt es neben der bekannten „p-q-Formel“ (mit $p = \frac{b}{a}$ und $q = \frac{c}{a}$) auch die so genannte „a-b-c-Formel“

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4\,ac}}{2a}$$

Setzen Sie diese Berechnung in Haskell zunächst mit **let-in** und dann mit **where** um. Dabei sollen x_1 und x_2 als 2-Tupel zurückgegeben werden.

b).

Seit der Veröffentlichung der Gaußschen Wochentagsformel haben verschiedene Mathematiker und Astronomen eine ganze Reihe vergleichbarer Formeln aufgestellt. Diese benötigen meist vorausberechnete Tabellen und verwenden komplizierte Sonderregeln, z.B. für Schaltjahre. Eine im Vergleich dazu sehr elegante Formel wurde 1972 von J. D. Robertson vorgestellt. Diese wird im Folgenden beschrieben. Gegeben sei ein beliebiges Datum (T,M,J) im Gregorianischen Kalender, wobei T den Tag des Monats beginnend mit 1, M den Monat des Jahres beginnend mit 1 und J das Jahr bezeichnet. Dann lässt sich der Wochentag W mit $0 \equiv$ Sonntag, $1 \equiv$ Montag, $\ldots$, $6 \equiv$ Samstag folgendermaßen berechnen:

$$W = \left(D + T + 77 + E + \left\lfloor \frac{B}{400} \right\rfloor - 2\left\lfloor \frac{B}{100} \right\rfloor\right) \; mod \; 7$$

Dabei gelte folgendes:

$$A = M + 10$$
$$B = \left\lfloor \frac{M-14}{12} \right\rfloor + J$$
$$C = A - 12\left\lfloor \frac{A}{13} \right\rfloor$$
$$D = \left\lfloor \frac{13 \cdot C - 1}{5} \right\rfloor$$
$$E = \left\lfloor \frac{5(B \; mod \; 100)}{4} \right\rfloor$$
$$\lfloor x \rceil = \begin{cases} \lfloor x \rfloor & x \geq 0 \\ \lceil x \rceil & x < 0 \end{cases}$$
$$\lfloor x \rfloor = max\{z \in \mathbb{Z} | z \leq x\}$$
$$\lceil x \rceil = min\{z \in \mathbb{Z} | z \geq x\}$$

Schreiben Sie eine Haskell-Funktion robertson :: Integer -> Integer -> Integer -> Integer, die gemäß der obigen Formel den Wochentag berechnet. Sie können sich mit der Hilfsfunktion *wochentag* den Wochentag als Text anzeigenlassen. Sie erhalten z.B. folgende Ausgaben:

- wochentag 19 6 1623 → "Montag"
- wochentag 19 1 2038 → "Dienstag"
- wochentag 30 4 1777 → "Mittwoch"

5. Logische Operatoren, Infix-Notation

Die booleschen Funktionen und sowie nicht seien wie folgt definiert:

```
und :: Bool -> Bool -> Bool
und = (&&)

nicht :: Bool -> Bool
nicht = not
```

Programmieren Sie die zweistelligen booleschen Funktionen oder, darausFolgt, genauDannWenn und entwederOder unter ausschließlicher Verwendung von und und nicht. Testen Sie Ihre Funktionen.

6. Rekursion über den natürlichen Zahlen

In den folgenden Teilaufgaben dürfen keine anderen arithmetischen Operationen als die beiden vordefinierten Funktionen succ und pred verwendet werden.

a).

Programmieren Sie eine Funktion `plus :: Integer -> Integer -> Integer` welche Summen natürlicher Zahlen berechnet. Nutzen Sie aus, dass `a + b = succ(pred a + b)` und `0 + a = a` gilt.

b).

Programmieren Sie eine Funktion `mal :: Integer -> Integer -> Integer` welche Produkte natürlicher Zahlen berechnet. Nutzen Sie aus, dass `a * b = b + (pred a * b)` und `0 * a = 0` bzw. `1 * a = a` gilt.

c).

Programmieren Sie eine Funktion `hoch :: Integer -> Integer -> Integer` welche Potenzen natürlicher Zahlen berechnet.

d).

Die Reihe plus, mal, hoch kann beliebig fortgesetzt werden. Programmieren Sie eine Funktion `meta :: Integer -> Integer -> Integer -> Integer` sodass `meta 0 a b$` der Summe `plus a b`, `meta 1 a b` dem Produkt `mal a b`, `meta 2 a b` der Potenz `hoch a b` usw. entspricht. Was berechnet `meta 3 a b`?

7. Datentypen

a).

Geben Sie für die folgenden Typdeklarationen die Namen und Typen der automatisch generierten Konstruktoren und Selektoren an:

```
data Ampelfarbe = Rot | Gelb | Gruen
data Menge = Stueck Integer | Kilogramm Double | Liter Double
data HausOderPostfach = Haus {
    strasse :: String,
    hausNummer :: Integer
  }
  | Postfach {
    postfachNummer :: Integer
  }
data Adresse = Adresse {
    hausOderPostfach :: HausOderPostfach,
    ort :: String,
    plz :: Integer
  }
```

b).

Geben Sie für den Datentyp Menge eine äquivalente Deklaration an, bei der automatisch Selektorfunktionen generiert werden. Wählen Sie sinnvolle Namen für die Selektorfunktionen.

c).

Geben Sie für die Datentypen HausOderPostfach und Adresse äquivalente Deklarationen an, bei denen nicht automatisch Selektoren generiert werden.

d).

Programmieren Sie aufbauend auf den Typdeklarationen aus Teilaufgabe 1 Selektorfunktionen mit den folgenden Signaturen:

```
strasseVonAdresse :: Adresse -> String
postfachNummerVonAdresse :: Adresse -> Integer
```

8. Pattern-Matching

Gegeben sind die folgenden Typdeklarationen:

```
data SI = SI String Integer
data SIListe = Leer | NichtLeer SI SIListe
data VielleichtSI = Nicht | Doch SI
```

a).

Bestimmen Sie die Typen der Variablen in den folgenden Mustern. Begründen Sie Ihre Antworten kurz.

```
loesche :: SI -> SIListe -> SIListe
loesche a Leer = error "nicht da"
loesche (SI b c) (NichtLeer (SI d e) f )
                 | b == d && c == e = f
                 | otherwise        = NichtLeer (SI d e) (loesche (SI b c) f )

ersetze :: SI -> SIListe -> SIListe
ersetze g Leer = Leer
ersetze h@(SI i _) (NichtLeer j@(SI k _) l) =
                 if i == k then NichtLeer h (ersetze h l)
                           else NichtLeer j (ersetze h l)

findeAnIdx :: Integer -> SIListe -> VielleichtSI
findeAnIdx _ Leer                          = Nicht
findeAnIdx m (NichtLeer n o) | m == 0      = Doch n
                             | otherwise = findeAnIdx ( pred m) o
```

Hinweis:
Das @ bindet den „zerlegten“ Datentyp an den vorstehenden Bezeichner, damit dieser nicht wieder zusammen gesetzt werden muss.

b).

Gegeben sind folgende Definitionen:

```
liste1 = NichtLeer (SI "Bert" 7)
         (NichtLeer (SI "Bianca" 9)
         (NichtLeer (SI "Bert" 7) Leer))
liste2 = NichtLeer (SI "Bert" 8)
         (NichtLeer (SI "Robert" 7) Leer)
liste3 = NichtLeer (SI "Robert" 8)
         (NichtLeer (SI "Robert" 7) Leer)
```

Geben Sie die Resultate folgender Funktionsanwendungen an. Begründen Sie Ihre Antworten kurz.
a)

```
loesche (SI "Bert" 7) liste1
```

b)

```
ersetze (SI "Robert" 9) liste2
```

c)

```
ersetze (SI "Robert" 9) liste3
```

d)

```
findeAnIdx 1 liste3
```

9. Arithmetische Ausdrücke

Wir betrachten einen rekursiven Datentyp Ausdruck, dessen Werte arithmetische Ausdrücke darstellen. Als Operationen sind Addition, Subtraktion und Multiplikation zugelassen, atomare Einheiten können Konstanten und Variablen sein. Die Deklaration von Ausdruck ist wie folgt:

```
data Ausdruck = Konstante Integer
  | Variable String
  | Summe Ausdruck Ausdruck
  | Differenz Ausdruck Ausdruck
  | Produkt Ausdruck Ausdruck
deriving (Show) -- ermoeglicht Ausgabe der Werte im GHCi
```

Der Ausdruck $x+(3-x)\cdot y$ wird beispielsweise repräsentiert durch:

```
Summe (Variable "x") (Produkt (Differenz (Konstante 3) (Variable "x"))
  (Variable "y"))
```

a).

Geben Sie die Repräsentationen folgender arithmetischer Ausdrücke an:
a) $-x$
b) $3\cdot x+y$
c) $3\cdot(x+y)$
d) $3+x\cdot y$

b).

Geben Sie die arithmetischen Ausdrücke an, die folgendermaßen repräsentiert werden:
a)

```
Produkt (Differenz (Variable "x") (Konstante 0)) (Variable "y")
```

b)

```
Differenz (Variable "x") (Produkt (Konstante 0) (Variable "y"))
```

c)

```
Summe (Produkt (Konstante 3) (Variable "z")) (Konstante 5)
```

c).

Programmieren Sie eine Haskell-Funktion ausdruckNachString :: Ausdruck -> String welche arithmetische Ausdrücke in Zeichenfolgen konvertiert. Die ausgegebenen Zeichenfolgen sollen die arithmetischen Ausdrücke in Haskell-Syntax darstellen. Es ist z.B. folgendes möglich:

```
ausdruckNachString (Differenz (Konstante 3) (Variable "x")) = "3 - x"
```

Hinweis: Zum Aneinanderhängen von Zeichenfolgen kann der Operator ++ verwendet werden.

```
(++) :: String -> String -> String
```

Ganzzahlen können durch Anwenden der Funktion show in ihre Zifferndarstellung konvertiert werden. Es ergibt sich beispielsweise

```
show 325 = "325"
```

10. Rekursion über Listen

Die folgenden Teilaufgaben sollen ohne Verwendung vordefinierter Bibliotheksfunktionen gelöst werden. Berücksichtigen Sie, dass die zu programmierenden Funktionen nicht für alle Eingaben sinnvoll definiert sind. In diesen Fällen sollen entsprechende Fehlermeldungen ausgegeben werden.

a).

Programmieren Sie eine Haskell-Funktion elementAnIdx :: Integer −> [el] −> el, die zu einem Index i und einer Liste das i-te Element dieser Liste bestimmt. Das erste Element einer nichtleeren Liste habe dabei den Index 0. Es ergibt sich zum Beispiel:
elementAnIdx 2 [3, 4, 5, 6] = 5

b).

Programmieren Sie eine Haskell-Funktion snoc :: [el] −> el −> [el], die an eine Liste ein weiteres Element anhängt. Es ergibt sich zum Beispiel:
snoc [1, 2, 3] 5 = [1, 2, 3, 5]

c).

Programmieren Sie eine Haskell-Funktion rueckwaerts :: [el] −> [el], welche eine Liste umdreht, die Elemente der Liste also in umgekehrter Reihenfolge zurück gibt.

d).

Programmieren Sie eine Haskell-Funktion praefix :: Integer −> [el] −> [el], die für eine gegebene Zahl n und eine Liste l die Liste der ersten n Elemente von l zurück gibt.

e).

Programmieren Sie eine Haskell-Funktion suffix :: Integer −> [el] −> [el], die für eine Zahl n und eine Liste l die Liste der letzten n Elemente von l zurückgibt.