STARKER START ins Studium

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Fachbereich Informatik und Mathematik
Institut für Informatik

Vorsemesterkurs Informatik

Sommersemester 2021

Aufgabenblatt Nr. 5a

Aufgabe 1 (Fehler in Haskell-Quelltext: Parsefehler)

Laden Sie von der Webseite des Vorkurses `http://vorkurs.informatik.uni-frankfurt.de` die Datei `blatt2-parse-fehlerA.hs` herunter und speichern Sie diese im Unterverzeichnis `vorkurs` Ihres Homeverzeichnisses.

Finden und beheben Sie alle Fehler im Quelltext. Gehen Sie dazu folgendermaßen vor: Starten Sie den GHCi und laden Sie die entsprechende Datei mit `:load blatt2-parse-fehlerA.hs` in den Interpreter. Sie erhalten die Fehlermeldung:

```
blatt2-parse-fehlerA.hs:4:10: parse error on input ‘)’
```

Diese besagt, dass der Fehler in der **4.** Zeile und der **10.** Spalte ist, und dass der GHCi nichts mit der Klammer „)“ anfangen kann.

Schauen Sie im Quelltext in der entsprechenden Zeile und Spalte nach:

```
f x = x+x)
```

Löschen Sie die überflüssige Klammer, speichern Sie die Quelltextdatei und laden Sie Datei erneut im GHCi usw.

Finden und korrigieren Sie auf diese Weise alle Fehler, bis der Quelltext vom GHCi ohne Fehler geladen werden kann. Versuchen Sie dabei die Fehler sinnvoll zu verbessern, d.h. nicht durch bloßes Auskommentieren der fehlerhaften Zeile.

Hinweise:

- Zwei der häufigsten Fehlerquellen in Haskell Quellcode sind *Parsefehler* und *Typfehler*.
- Die Datei `blatt2-parse-fehlerA.hs` enthält hauptsächlich *Parsefehler*. Das sind syntaktische Fehler im Quelltext, wie z.B. nicht erlaubte oder vergessene Zeichen, falsche Einrückung, u.a.
- Die angegebene Zeile und Spalte der Fehlermeldung muss nicht immer genau passen, der Fehler kann sich z.B. auch am Ende der darüber liegenden Zeile befinden.

Aufgabe 2 (Einfache Funktionen in Haskell definieren)

Legen Sie in einem Editor eine Haskell-Quelltextdatei (die Datei-Endung muss `.hs` sein) an und implementieren Sie dort die folgenden Funktionen in Haskell:

a) Eine Funktion `ignoriere :: Integer -> Integer`, die zwar eine Zahl als Eingabe nimmt, aber stets als Ergebnis die Zahl 0 liefert.

Tipp: Da die Funktion einen Parameter erwartet, wird der Quelltext so anfangen:

```
ignoriere :: Integer -> Integer
ignoriere x =
```

Lösung

```
ignoriere :: Integer -> Integer
ignoriere x = 0
```

b) Eine Funktion `minusEins :: Integer -> Integer`, die eine Zahl als Eingabe nimmt und 1 von der Zahl abzieht.

Lösung

```
minusEins :: Integer -> Integer
minusEins x = x-1
```

c) Eine Funktion `minGanz :: Integer -> Integer -> Integer`, die *zwei* Zahlen als Eingabe erhält und die kleinere der beiden Zahlen als Ergebnis liefert.

Tipps:

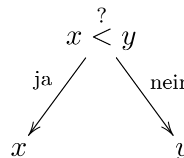
- Da die Funktion zwei Eingaben erwartet, wird der Quelltext so anfangen:

```
minGanz :: Integer -> Integer -> Integer
minGanz x y =
```

- Verwenden Sie einen `if-then-else`-Ausdruck, da sich das Verhalten der Funktion doch Ausdrücken lässt als:

Wenn x kleiner als y ist,
dann gib x zurück und
ansonsten gib y zurück.

Oder dargestellt als Entscheidungsbaum:



Lösung

```
minGanz :: Integer -> Integer -> Integer
minGanz x y = if x < y then x else y
```

d) Eine Funktion `max3Ganz :: Integer -> Integer -> Integer -> Integer`, die *drei* Zahlen als Eingabe erhält und die größte der Zahlen als Ergebnis liefert.

Tipps:

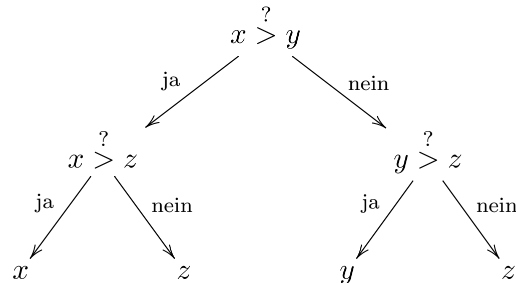
– Da die Funktion drei Eingaben erwartet, wird der Quelltext so anfangen:

```
max3Ganz :: Integer -> Integer -> Integer -> Integer
max3Ganz x y z =
```

– Verwenden Sie *verschachtelte* `if-then-else`-Ausdrücke. Das Verhalten der Funktion lässt sich ausdrücken als:

Wenn x größer als y ist,
dann
(*wenn* x auch größer als z,
dann gib x zurück,
sonst gib z zurück)
ansonsten
(wenn y größer z ist,
dann gib y zurück,
sonst gib z zurück)

Oder dargestellt als Entscheidungsbaum:



Lösung

```
max3Ganz :: Integer -> Integer -> Integer -> Integer
max3Ganz x y z = if x > y then
                   if x > z then x else z
                 else
                   if y > z then y else z
```

e) Eine Funktion `min3Ganz :: Integer -> Integer -> Integer -> Integer`, die *drei* Zahlen als Eingabe erhält und die kleinste der Zahlen als Ergebnis liefert.

Tipp: Sie dürfen `minGanz` aufrufen, um das Minimum von zwei Zahlen zu berechnen.

Lösung

```
min3Ganz :: Integer -> Integer -> Integer -> Integer
min3Ganz x y z = minGanz (minGanz x y) z
```

f) Eine Funktion `median :: Integer -> Integer -> Integer -> Integer`, die drei Zahlen als Eingaben erhält und den Median der drei Zahlen liefert, d.h. die Zahl, die weder die größte noch die kleinste ist. Z.B. ergibt `median 1 7 5` die Zahl `5` und `median 2 8 1` die Zahl `2`.

Lösung

```
median :: Integer -> Integer -> Integer -> Integer
median x y z = if x > y
                 then if y > z
                        then y
                        else if x > z
                               then z
                               else x
                 else if x > z
                        then x
                        else if y > z
                               then z
                               else y
```

Aufgabe 3 (Fehler in Haskell-Quelltext: Typfehler)

Laden Sie von der Webseite des Vorkurses `http://vorkurs.informatik.uni-frankfurt.de` die Datei `blatt2-typ-fehlerA.hs` herunter und speichern Sie diese im Unterverzeichnis `vorkurs` Ihres Homeverzeichnisses.

Laden Sie die Datei in den GHCi. Sie erhalten eine Liste von Fehlermeldungen. Der erste Fehler ist:

```
blatt2-typ-fehlerA.hs:4:10:
    Couldn't match expected type 'Bool -> t0' with actual type 'Bool'
    The function 'True' is applied to one argument,
    but its type 'Bool' has none
    In the expression: True False
    In an equation for 'test_1': test_1 = True False
```

Der Fehler ist daher in Zeile **4** und Spalte **10**. Die Fehlermeldung verrät aber noch mehr: Man darf `True` nicht auf `False` anwenden, da `True` keine weiteren Argumente verarbeiten kann. Genauer noch: `True` ist vom Typ `Bool`, und daher keine Funktion (Funktionstypen haben schließlich einen Pfeil `->` im Typen).

Verbessern Sie den Fehler, indem Sie Zeile 4 durch

```
test_1 = True
```

ersetzen.

Verbessern Sie analog alle weiteren Typfehler in der Datei und machen Sie sich dabei stets klar, warum das Programm an der entsprechenden Stelle einen Fehler hat.

Aufgabe 4 (Pfefferdieb als Funktion)

Wir kennen bereits das Programm zum Rätsel vom Pfefferdieb vom letzten Aufgabenblatt:

```
hutmacher   = undefined
schnapphase = undefined
haselmaus   = undefined

genau_einer =
    (hutmacher && not schnapphase && not haselmaus)
 || (not hutmacher &&  schnapphase && not haselmaus)
 || (not hutmacher && not schnapphase &&  haselmaus)

aussage1 = schnapphase || (not hutmacher)
aussage2 = hutmacher   || (not haselmaus)

raetsel = genau_einer && aussage1 && aussage2
```

Das Finden des Pfefferdiebs war etwas mühsam, da wir die Werte von `hutmacher`, `schnapphase` und `haselmaus` immer im Quelltext ändern und dann neu im GHCi laden mussten. Besser wäre es, wenn `raetsel` eine *Funktion* wäre, welche die Wahrheitswerte für `hutmacher`, `schnapphase` und `haselmaus` als Eingaben erhält, d.h.

```
raetsel hutmacher schnapphase haselmaus = ...
```

Dann können wir uns die drei ersten Zeilen des Programms sparen, und direkt im GHCi z.B. testen:

```
Prelude> raetsel True False False
```

Passen Sie das Programm zum Pfefferdieb an, sodass `raetsel` eine Funktion vom Typ `Bool -> Bool -> Bool -> Bool` ist, die als Eingaben die Wahrheitswerte der drei Verdächtigen erhält und genau dann `True` liefert, wenn der Pfefferdieb enttarnt ist.

Tipps:

- Die drei ersten Zeilen

  ```
  hutmacher   = undefined
  schnapphase = undefined
  haselmaus   = undefined
  ```

  können Sie aus dem Programm löschen

- Sie müssen auch `genau_einer`, `aussage1`, `aussage2` so modifzieren, dass sie Funktionen (mit mehr als 0 Argumenten) werden.

Lösung

```
genau_einer hutmacher schnapphase haselmaus =
    (hutmacher && not schnapphase && not haselmaus)
 || (not hutmacher &&  schnapphase && not haselmaus)
 || (not hutmacher && not schnapphase &&  haselmaus)

aussage1 hutmacher schnapphase =
    schnapphase || (not hutmacher)

aussage2 hutmacher haselmaus =
    hutmacher   || (not haselmaus)

raetsel hutmacher schnapphase haselmaus  = (genau_einer hutmacher schnapphase haselmaus)
                                            && aussage1 hutmacher schnapphase
                                            && aussage2 hutmacher haselmaus
```

Danach kann man testen:

```
*Main> raetsel False True False
True
*Main> raetsel False False True
False
```

Aufgabe 5 (Rekursion)

In dieser Aufgabe sollen Sie eine Funktion `summeGeradeZahlen :: Integer -> Integer` implementieren, die

1. eine gerade positive Zahl n erwartet und die Summe $2 + 4 + 6 \ldots + n$ *rekursiv* berechnet.
2. Falls die eingegebene Zahl nicht gerade und positiv ist, so soll das Ergebnis stets 0 sein.

Teil 2 können Sie implementieren, indem Sie mit einer `if-then-else`-Abfrage zunächst prüfen, ob die Zahl gerade[1] und positiv ist.

Die eigentliche Summenberechnung soll nach dem Prinzip der Rekursion erfolgen:

[1] Die Funktion `even` testet, ob eine Zahl gerade ist.

- Rekursionsanfang: Wenn die eingegebene Zahl die 2 ist, dann muss nicht mehr summiert werden, und die 2 wird als Ergebnis geliefert.
- Rekursionsschritt: Wir lassen die Summe $2+4+\ldots+n-2$ *rekursiv* berechnen, und müssen dann nur n dazu addieren.

Lösung

```
summeGeradeZahlen n =
  if even n && n > 0 then
     -- Rekursionsanfang:
     if n == 2 then 2
     else
     -- Rekursionsschritt
       (summeGeradeZahlen (n-2)) + n
  else 0 -- keine gerade Zahl
```