STARKER START ins Studium

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Fachbereich Informatik und Mathematik
Institut für Informatik

Vorsemesterkurs Informatik

Sommersemester 2020

Aufgabenblatt Nr. 5B

Aufgabe 1 (Fehler in Haskell-Quelltext: Parsefehler)

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

Identifizieren und verbessern Sie alle Fehler in der Quelltextdatei, so dass diese sich ohne Fehlermeldung im GHCi laden lässt.

Gehen Sie dazu folgendermaßen vor: Starten Sie den GHCi und laden Sie die entsprechende Datei in den Interpreter. Lesen Sie die Fehlermeldung, öffnen Sie anschließend die Datei in einem Editor und verbessern Sie den Fehler. Laden Sie anschließend die Datei erneut im GHCi, um den nächsten Fehler zu finden. Versuchen Sie die Fehler sinnvoll zu verbessern, d.h. nicht durch bloßes Auskommentieren der fehlerhaften Zeile.

Hinweis: Zwei der häufigsten Fehlerquellen in Haskell Quellcode sind *Parsefehler* und *Typfehler*.

Die Datei `blatt2-parse-fehlerB.hs` enthält hauptsächlich *Parsefehler*. Das sind syntaktische Fehler im Quelltext, wie z.B. nicht erlaubte oder vergessene Zeichen, falsche Einrückung, u.a.

Aufgabe 2 (Einfache Funktionen in Haskell definieren)

a) Legen Sie in einem Editor eine Haskell-Quelltextdatei an und implementieren Sie dort die folgenden Funktionen (mit expliziten Typangaben, wie in der Aufgabenstellung vorgegeben) in Haskell:

- Eine Funktion `inc :: Integer -> Integer`, die eine gegebene ganze Zahl (vom Typ `Integer`) um 1 erhöht.
- Eine Funktion `ver_n_fache :: Integer -> Integer -> Integer`, die zwei ganze Zahlen `a` und `n` erhält und die Zahl `a` n-mal aufaddiert.
- Eine Funktion `sum_ganz :: Integer -> Integer -> Integer`, die zwei ganze Zahlen addiert.
- Eine Funktion `abs_ganz :: Integer -> Integer`, die den Betrag einer ganzen Zahl berechnet.
- Eine Funktion `max_ganz :: Integer -> Integer -> Integer`, die das Maximum zweier ganzer Zahlen berechnet.
- Eine Funktion `max5_ganz :: Integer -> Integer -> Integer -> Integer -> Integer -> Integer`, die das Maximum fünf ganzer Zahlen berechnet.

Lösung

```
inc :: Integer -> Integer
inc x = x + 1

ver_n_fache :: Integer -> Integer -> Integer
ver_n_fache a n = a * n

sum_ganz :: Integer -> Integer -> Integer
sum_ganz x y = x + y

abs_ganz :: Integer -> Integer
abs_ganz x = if x < 0 then -1 * x else x

max_ganz :: Integer -> Integer -> Integer
max_ganz x y = if x < y then y else x
max5_ganz :: Integer -> Integer -> Integer -> Integer -> Integer -> Integer
max5_ganz x1 x2 x3 x4 x5 = max_ganz (max_ganz (max_ganz x1 x2) (max_ganz x3 x4)) x5
```

b) Vergleichen Sie die Typen von `abs_ganz` und `max_ganz` mit den Typen der vordefinierten Funktionen `abs` und `max`. Wo liegen die Unterschiede?

Löschen Sie die expliziten Typangaben von `abs_ganz` sowie `max_ganz` und vergleichen Sie nun deren Typen erneut mit denen der vordefinierten Funktionen `abs`, `max`.

Lösung

Die Typen der vordefinierten Funktionen sind allgemeiner (polymorph):

```
*Main> :t abs
abs :: (Num a) => a -> a
*Main> :t max
max :: (Ord a) => a -> a -> a
```

D.h. sie sind für mehr Typen als nur `Interger` verwendbar:

```
*Main> abs (-1/2)
0.5 :: Double
*Main> abs_ganz (-1/2)

    No instance for (Fractional Integer)
      arising from a use of ‘/’ at <interactive>:1:11-13
    Possible fix: add an instance declaration for (Fractional Integer)
    In the expression: 1 / 2
    In the first argument of ‘abs_ganz’, namely ‘(- 1 / 2)’
    In the expression: abs_ganz (- 1 / 2)
```

Die Typklassenbeschränkungen geben zusätzliche Bedingungen an die Typen an: `abs` ist nur für Typen verwendbar die Instanzen der Typklasse `Num` sind (d.h. *abs* ist nur auf Zahlen anwendbar) und `max` nur für Typen die Instanzen der Typklasse `Ord` sind (das sind Typen auf denen eine Ordnungsrelation definiert ist).
Löscht man die Typangaben der Funktionen, werden die Funktionstypen vom GHCi hergeleitet: Er berechnet für alle Funktionen der Aufgabe einen allgemeineren Typ als den Angegeben. Z.B. für `inc :: (Num a) => a -> a`. D.h. die Typangabe kann beim programmieren in Haskell weggelassen werden.

Aufgabe 3 (Fehler in Haskell-Quelltext: Typfehler)

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

Identifizieren und verbessern Sie (sinnvoll, d.h. nicht durch Auskommentieren) alle Fehler in den beiden Quelltextdateien, so dass diese sich ohne Fehlermeldung im GHCi laden lassen.

Hinweis: Die Datei `blatt2-typ-fehlerB.hs` enthält vor allem *Typfehler*. Solche Fehler treten auf, wenn Operatoren oder Funktionen auf Datentypen angewendet werden, für die sie nicht definiert sind. Beispielsweise ergibt `'A' + 'B'` einen Typfehler, da die Operation `+` nicht auf Zeichen die `'A'`,`'B'` anwendbar ist.

Aufgabe 4 (Funktionen in Haskell)

Legen Sie eine Haskell-Quelltext-Datei an und implementieren Sie die folgenden Funktionen. Testen Sie die Funktionen ausgiebig mit sinnvollen Werten.

a) Eine Funktion, die zwei ganze Zahlen `x` und `y` als Eingaben erhält und die Summe der beiden Zahlen liefert, falls `x` größer als `y` ist und anderenfalls das Produkt der beiden Zahlen als Ergebnis liefert.

Lösung

```
fun1 x y = if x > y then x+y else x*y
```

b) Eine Funktion, die eine ganze Zahl als Eingabe erhält und die Zahl quadriert, falls die Zahl *ungerade* ist und anderenfalls die Zahl verzehnfacht.

Lösung

```
fun2 x = if odd x then x*x else 10*x
```

c) Eine Funktion, die eine ganze Zahl als Eingabe erhält und

- 1 liefert, falls die Zahl negativ und ungerade ist.
- 2 liefert, falls die Zahl positiv und gerade oder positiv und größer als 200 ist.
- 3 liefert, falls die Zahl positiv, ungerade und kleiner als 100 ist.
- 0 in allen anderen Fällen liefert.

Lösung

Z.B.:

```
fun3 x = if x < 0 && odd x then 1
         else if (x > 0 && even x) || (x > 200) then 2
              else if x > 0 && x < 100 && odd x then 3
                   else 0
```

d) Eine Funktion, die eine ganze Zahl `x` und einen Booleschen Wert `b` als Eingaben erhält und

- `x` verdoppelt, falls `x` eine positive Zahl ist und `b` falsch ist.
- `x` verdreifacht, falls `x` durch 100 teilbar ist und `b` wahr ist.
- `x` in allen anderen Fällen zurückliefert.

Lösung

```
fun4 x b = if x > 0 && (not b) then 2*x
          else if (x `mod` 100 == 0) && b then 3*x
               else x
```

e) Eine Funktion `xor`, die das *exklusive-Oder* zweier boolescher Werte entsprechend der folgenden Wertetabelle berechnet:

`a`	`b`	Wert von `xor a b`
`False`	`False`	`False`
`False`	`True`	`True`
`True`	`False`	`True`
`True`	`True`	`False`

Lösung

Z.B.

```
xor a b = (a || b) && not (a && b)
```

oder

```
xor a b = if a then not b else b
```

Aufgabe 5 (Darts)

In der unteren Abbildung sind die Regeln des Darts-Spieles angegeben. Implementieren Sie in Haskell mit Hilfe der Fallunterscheidung eine Funktion `einWurf`, die den neuen Punktestand nach einem Wurf im Darts-Spiel berechnet. Die Funktion `einWurf` sollte sieben formale Parameter haben: Das Segment (Zahl zwischen 1 und 25) und die Wertigkeit (Zahl zwischen 1 und 3) für jeden der drei geworfenen Dart-Pfeile, sowie den Punktestand vor dem Wurf. Als Ergebnis soll sie den Punktestand nach dem Wurf liefern. Beachten Sie, dass die spezifierten Regeln eingehalten werden und dass es auch Würfe gibt, die die Scheibe verfehlen (diese sollten Sie irgendwie repräsentieren).

Lösung

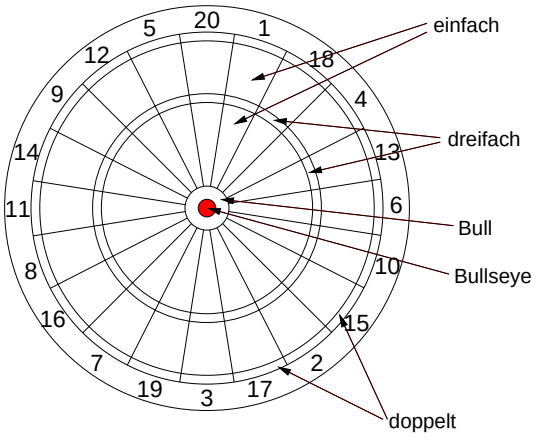
```
-- si: getroffenes Segment des i-ten Pfeils (0 wenn Scheibe verfehlt)
-- wi: Wertigkeit des i-ten Pfeils (0 wenn Scheibe verfehlt)
-- p:  Punktestand vor dem Wurf


-- Beim Ueberwerfen muss man aufpassen:
--  1 Punkt darf nicht verbleiben, da man anschliessend kein Doppel zu Sieg mehr
--  werfen kann

einWurf s1 w1 s2 w2 s3 w3 p =
 if    ((p-s1*w1 == 0) && (w1==2))             -- Doppel zum Sieg im ersten Wurf
    || ((p-s1*w1-s2*w2 == 0) && (w2==2))       -- Doppel zum Sieg im zweiten Wurf
    || ((p-s1*w1-s2*w2-s3*w3 == 0) && (w3==2)) -- Doppel zum Sieg im dritten Wurf
 then 0
 else  if   (p-s1*w1 <= 1)                    -- Ueberworfen im ersten Wurf
         || (p-s1*w1-s2*w2 <= 1)              -- Ueberworfen im zweiten Wurf
         || (p-s1*w1-s2*w2-s3*w3 <= 1)        -- Ueberworfen im dritten Wurf
       then p
       else p-s1*w1-s2*w2-s3*w3
```

Das Dart-Board ist in 20 Zahlensegmente unterteilt. Die Felder zählen jeweils soviel, wie außen am entsprechenden Segment notiert ist. Ausnahmen sind die schmalen Kreise. Im äußeren Kreis zählen die Felder das Doppelte, im mittleren Kreis das Dreifache der entsprechenden Zahl. Der Mittelpunkt des Boards, das Bullseye, zählt 50 Punkte (Doppel 25), der Ring darum herum (Bull) 25 Punkte.

Das Spiel ist beendet, wenn der Spieler mit einem Doppel (äußerer Kreis oder Bullseye) genau 0 Punkte erreicht, d.h. mit einem Dart im Doppel-Ring bzw. Bullseye muss die Punktzahl genau auf Null gebracht werden.
Der Spieler beginnt bei 501 Punkten; ein Wurf besteht aus drei nacheinander geworfenen Darts. Die erzielten Punkte werden jedesmal von der verbleibenden Summe abgezogen. Es dürfen nur diejenigen Darts gezählt werden, die im Board steckenbleiben. Erzielt der Spieler zu viele Punkte, sodass er nicht mehr mit einem Doppel abschließen kann, so werden alle drei Darts dieses Wurfes nicht gezählt; der Spieler bleibt auf demselben Rest wie vor dem Wurf.



Aufgabe 6 (Rekursion)

Siggi Sparer eröffnet im Jahr 1 (am 1. Januar) ein Sparkonto bei der Frankfurter Lumbe&Labbe-Bank und zahlt direkt ein Startkapital von 800 € ein. Siggi zahlt zudem am 1. Januar jedes weiteren Jahres 400 € ein. In den ersten 20 Jahren wird das Guthaben mit jeweils 3% pro Jahr verzinst, danach mit 4% pro Jahr. Implementieren Sie in Haskell eine *rekursive* Funktion, die das Guthaben von Siggis Konto für den 1. Januar des n. Jahres berechnet.

Lösung

```
guthaben 1    = 800
guthaben jahr = let guthaben_davor = guthaben (jahr-1)
                in if jahr > 20 then ((guthaben_davor)*1.04)+400
                   else richtigRunden ((guthaben_davor*1.03)+400)

richtigRunden:: Double -> Double
richtigRunden x = (fromInteger ( round (x*100))) / 100
```