Hangman-AI in Haskell

Joachim Breitner

10.7.2006

Zusammenfassung

In meinem Informatik-II-Tutorium habe ich heute (mangels besseren Themen) gemeinsam mit meien Tutoranten ein Programm geschrieben, das relativ gut Hangman-Spiele, auch bekannt als Galgenmännchen, lösen können sollte.

Übersicht

Die Logik des Programme ist folgende: Wir haben eine Liste von möglichen Wörtern. Aus dieser wählen wir jene aus, die nach dem aktuellen Wissenstandt (dem „Pattern“) in Frage kommen und schauen, welche noch nicht gefragte Buchstabe kommt am häufigsten vor, und den Raten wir dann.

Die Worliste

Die Wortliste habe ich auf meinem Debian-Sytem wie folgt generiert:

```
cat /usr/share/dict/words |
        iconv -f latin1 -t utf8 |
        perl -n -p -e 's/ä/ae/ig; s/ö/oe/ig; s/ü/ue/ig;s/ß/ss/g;' > words
```

Der Code

Erst einmal laden wir einige nützliche Funktionen

```
2  import IO
3  import Char
4  import List
```

Dann brauchen wir eine Funktion, die schaut, ob ein gegebenes Wort (3. Parameter) noch in das Pattern (2. Parameter) passt, wobei Buchstaben, die im 1. Paramter vorkommen, nicht mehr hinzukommen dürfen.

```
5  match seen [] [] = True
6  match seen []  _ = False
7  match seen _  [] = False
8  match seen ('_':ps) (b:ws) | b `elem` seen = False — Der Unterstrich ist der Platzhalter
9                             | otherwise     = match seen ps ws
10 match seen (a:ps) (b:ws) = a == b && match seen ps ws
```

Eine weitere Funktion durchsucht die Worliste nach den Wörtern, die zum Pattern passen.

```
11 myfilter pat list seen = filter (match seen pat) list
```

`countb` ist eine Abkürzung für `lenght $ filter (==b)`, die hoffentlich etwas schneller ist.

```
12 countb _ [] = 0
13 countb b (x:xs) | b == x    = 1 + countb b xs
14                 | otherwise =     countb b xs
```

Diese Funktion ist aus dem Übungsbetrieb bekannt, hier jedoch etwas effizienter implementiert, da die rekursive Variante, wie sie auf Abgaben vorkam, bei unserer Wortliste zu einem Stack Overflow führte. Das Ergebnis ist eine Liste von Tupeln mit der Häufigkeit des Buchstabens und dem Buchstaben selbst.

```
15 count stat text seen = [ (countb b text, b)
16                                 | b ← ['A'..'Z'],
17                                   b `notElem` seen ]
```

Hier ist die Hauptfunktion, welche rekursiv für jeden Zug ausgeführt wird. Zuerst fragen wir nach dem Pattern, schauen, ob es überhaupt zum vorherigen passt (um ggf. nochmal zu fragen). Dann verkleinern wir die Worliste auf die noch möglichen Wörter. Bliebt nix mehr übrig, geben wir auf, kommt nur noch ein Wort in Frage, so geben wir dieses aus, ansonsten nehmen wir den häufigsten Buchstaben, und rufen uns selbst wieder rekursiv auf.

```
18 denken liste seen' oldpat = do
19         putStrLn $ "Wie sieht das Pattern denn grad aus?"
20         pat' ← getLine
21         let pat = map toUpper pat' — nachsichtige Eingabe
22         if not $ null oldpat || match [] oldpat pat  — oldpat ist am Anfang leer
23                 then do putStrLn "Nein, ich esse mein Pattern nicht, mein Pattern ess ich nicht!"
24                         denken liste seen' oldpat — Wiederholung der Eingabe
25                 else do
26                   — Zeichen im Pattern sind „gesehen“
27         let seen = seen' ++ [ b | b ← pat, b /= '_', b `notElem` seen']
28         let filtered = myfilter pat liste seen
29         if null filtered — keine Ahnung mehr
30                 then    putStrLn "Ich gebe auf..."
31                 else do
32         if all (/='_') pat' — Keine unbekannten mehr
33                 then putStr "Juhuuuh.\n"
34                 else do
35         if null $ tail filtered — nur noch ein Kandidat
36                 then    putStrLn $ "Ich löse: "++(head filtered)
37                 else do
38         if length filtered < 5 — kleine Ausgabe bei wenig Kandidaten
39                 then putStr $ "Hmm, eigentlich nur: \n"++(unlines $ map ("    "++) filtered)
40                 else return ()
41         let (_,guess) = last $ sort $ count [] (concat filtered) seen
42         putStrLn $ "Ich kaufe ein "++(show guess)
43         denken filtered (guess:seen') pat — und weiterraten
```

Das Hauptprogramm läd die Wortlistendatei und macht daraus eine Liste von Wörtern aus Großbuchstaben, um dann obige Funktion ohne Pattern und gesehenen Buchstaben aufruft.

```
44 main = do
45         wortliste ← readFile "words"
46         let woerter = map (map toUpper) $ lines wortliste
47         denken woerter [] []
```

Fazit

Das Ergebnis war zufriedenstellend. Das Programm war zwar nicht schnell, aber die Wartezeite nwaren nicht zu lange (dramaturgisch eher besser), und die Rategenauigkeit war gut. So brauchte das versammelte Tutorium für „OMEGA“ 10 Buchstaben, das Programm nur 7...