Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Dokumentation
Parametric Programming
(Programmiersprache: Haskell)

Erstellung und Benutzung eines assoziativen Indexes für Programmtexte

Christian Bunk
Alexander Miller

Abgabedatum: 19.07.2011

Prof. Dr. -Ing. Horst Hansen

Inhaltsverzeichnis

1. Motivation und Problemstellung

Rein funktionale Programmiersprachen bieten eine besondere Möglichkeit zur Implementierung von Lösungen für komplexe mathematische Problemstellungen. Haskell gehört zu der Klasse von rein funktionalen Programmiersprachen, welche ebenfalls das effiziente Arbeiten mit mathematischem Hintergrund ermöglicht. Im Laufe der Lehrveranstaltung "Parametric Programming" wurden die einzelnen Möglichkeiten von Haskell vorgestellt und an kleinen Beispielen erläutert. Als Prüfungsleistung müssen zwei Projekte erfolgreich realisiert werden, wobei das in der Vorlesung erworbene Wissen praktisch angewandt werden sollte.

Bei der vorliegenden Ausarbeitung handelt es sich um eine Dokumentation zum zweiten Projekt. Wie in der Aufgabenstellung definiert (Anhang A.1) soll ein Programm zur "Erstellung und Benutzung eines assoziativen Indexes für Programmtexte" modelliert und entwickelt werden. Da die funktionale Programmiersprache Haskell für die Realisierung verwendet werden sollte, wurden im Laufe des Projektes ausschließlich die Standardbibliotheken verwendet.

2. Beschreibung der Programmstruktur

Wie in der Aufgabenstellung vordefiniert wurde die Implementierung des Programms, welches die vorgegebene Problemstellung effektiv löst, mit einer rein funktionalen Programmiersprache Haskell realisiert. Aus diesem Grund besteht das Programm aus mehreren Funktionen die eigentliche Funktionalität des Programms beinhalten. Darüber hinaus wurden nur die Standard-Bibliotheken der Sprache Haskell sowie GHC verwendet. Die interne Datenhaltung für den erstellten Index erfolgt ausschließlich durch die Datenstrukturen der Standard-Bibliotheken. Wie die praktische Umsetzung im Detail realisiert wurde, kann der Beschreibung der einzelnen Funktionen entnommen werden. Die Datenstruktur für den ermittelten Index wird wie folgt realisiert: `[( Wort , [( File , [ Int ])])]`

2.1. Main

Die Datei Main.hs beinhaltet den Quellcode, welcher den Ablauf des Programms steuert und bei verschiedenen Übergabeparametern die entsprechende Aktionen ausführt. Zuerst werden die Kommandozeilenparametern eingelesen und mittels `if ... then ... else` Abfragen ausgewertet, sodass sogar eine Kombination der einzelnen Optionen möglich ist. Erstellung des Indexes sowie die anschließende Ausgabe in einer Datei oder am Terminal sind weitere Aufgaben, die von diesem Bereich des Programms aufgerufen werden.

2.2. Cmdlineargs

Laut der Aufgabenstellung erfolgt die Steuerung der Programms ausschließlich über die Kommandozeile. Dabei wurden alle IST und SOLL Funktionalitäten implementiert, sodass das resultierende Programm mit folgenden Parametern aufgerufen werden kann:

<program> <options> <outputfile> <inputfile>*
<program> : Program name
<options> :

-p Print index list on data terminal
-i Create an index
-q=<word> Print all indexes for word <word> on data termnal
-s=<prefixterm> Print indexes for all words with prefix term <prefixterm>
-t=<filename> Print indexes for words founded in file <filename>
-c Number of words in current operation

Das Einlesen der einzelnen Parametern erfolgt mit der zur Verfügung stehenden Funktion `args <- getArgs`. Die übergebenen Optionen werden von Ein- bzw. Ausgabedateien getrennt, sodass durch den Aufruf von `getOptions args` und `getInputFiles args` und `getOutputFile args` alle erforderlichen Informationen in dazu vorgesehenen Datenstrukturen zur Verfügung stehen. Im Falle einer fehlerhaften Eingabe wird der Benutzer auf die richtige Eingabe der Parameter hingewiesen.

2.3. Index

Für die Erstellung des Indexes wurde auf die Datenstruktur Map, welche von GHC implementiert und geliefert wird, vollständig verzichtet. Für diesen Zweck wird die Datenstruktur Map mittels einfachen Datentypen abgebildet, sodass die erforderlichen Informationen in folgender Art und Weise aufbewahrt werden: `[( Wort , [( File , [ Int ])])]` . Dabei handelt es sich um eine Liste von Tupeln, wobei als erstes Wert das Wort und als zweites eine Liste von Tupeln gespeichert wird. Die zweite Liste von Tupeln besteht wiederum aus einem String, welches die Dateiname erfasst, sowie einer Integer-Liste. Diese Liste repräsentiert alle Zeilennummern in welchen das Wort vorkommt. Somit erfolgt die Realisierung der Funktion `index :: [( Text , File )] -> [( Wort , [( File , [ Int ])])]`, welche den Index erstellt und für weiteren Ablauf des Programms bereitstellt.

2.4. Parser

Dieser Programmbereich realisiert die Funktionalität eines Parsers für eine bereits erstellte Index-Datei. Hierzu wird die Datei eingelesen und mit Hilfe der Funktionen `parseWort :: Zeile -> (Wort , Zeile )` und `parseFile :: Zeile -> (File , Zeile )` und `parseLines :: Zeile -> [ Int ]` die jeweiligen Worte, Dateinamen sowie die einzelnen Indizien ermittelt und zur Verfügung gestellt. Die Funktion `parse :: String -> [( Wort , [( File , [ Int ])])]` liefert die vollständige Datenstruktur eines Indexes und kann auf die entsprechenden Index-Files angewendet werden.

A. Anhang

A.1. Aufgabenstellung

2. Belegaufgabe

Hochschule für Technik und Wirtschaft Berlin
University of Applied Sciences

Prog. mit parametrisierten Datentypen	Sommersemester 2011	Dozent: Horst Hansen
Ausgabe: 14.6.2011	Abgabe Gruppe 1: 12.7.2011, Gruppe 2: 19.7.2011	

Lernziele:
Mit der Lösung dieser Aufgabe sollen sie zeigen, daß Sie in der Lage sind, Anwendungen funktional zu strukturieren und in der rein funktionalen Programmiersprache Haskell zu implementieren.

Spezifische Ziele:

- Verwendung parametrisierter Datentypen beim Entwurf von Programmen
- Benutzen der Standardbibliotheken von Haskell
- Benutzen der Standardbibliotheken von GHC
- Benutzen der Ein-/Ausgabefunktionen von Haskell
- Dokumentation von Programmen

Aufgabe: Erstellung und Benutzung eines assoziativen Indexes für Texte

Es soll ein Programm zum Indizieren von Texten erstellt werden. Es wird für jedes Wort ein Index erstellt, der angibt, in welcher Zeile das Wort auftritt. Der erstellte Index wird in vorgegebener Form (siehe unten) in eine Textdatei und/oder am Terminal ausgegeben. Anschließend kann der Index zum Beantworten von Fragen verwendet werden.

In den folgenden Abschnitten werden die Anforderungen an das zu erstellende Programm aufgeführt. Alle Anforderungen, bei denen nichts anderes angegeben ist, sind *Muß-Kriterien*. Blau gedruckte Anforderungen sind *Soll-Kriterien*.

Funktionale Anforderungen

Um Programme mit vergleichbaren Ergebnissen zu erhalten, gelten verbindlich die folgenden Definitionen für die Lösung der Aufgabe:

- Das Programm indiziert die Wörter einer oder mehrerer Eingabedatei(en) und gibt den errechneten Index in die Ausgabedatei und optional am Terminal aus.
- Wörter werden beim Erstellen der Indexes in ihrer Schreibweise nicht verändert.
- Das Programm verwendet den erstellten Index zum Beantworten von Anfragen des Benutzers.
- Ein Wort ist eine zusammenhängende Folge von Zeichen, die mit einem Unterstrich oder einem Buchstaben beginnt und anschließend **deutsche** Buchstaben, Ziffern, Bindestriche oder Unterstriche enthält.
  Oder als regulärer Ausdruck: `([A-Za-z_] | Umlaute) ([A-Za-z0-9]|-|_| Umlaute)*` . `Umlaute` sind die deutschen Umlaute und ß.
- Alle anderen Zeichen sind *Trennzeichen*, d.h. sie beenden ein Wort.
- Die lexikografische Sortierung der Wörter behandelt die Umlaute so: ä wird als ae betrachtet, usw., ß als ss!

A. Anhang

Anforderungen an die Benutzungsschnittstelle

- Die Steuerung des Programms erfolgt ausschließlich über die Kommandozeile.
- Das Programm soll mit den folgenden Kommandozeilenparametern gestartet werden:

  `<program> <option>* <outputfile> <inputfile>*`
  `<program>` : Programmname
  `<option>` :

  - `-p` Ausgabe der Indexliste am Terminal
  - `-i` Erstellen des Indexes
  - `-c` Ausgabe der Anzahl der Wörter in der jeweiligen Antwort am Terminal
  - `-q=<wort>` : Ausgeben des vollständigen Indexes zum Wort `wort` am Terminal
  - `-s=<wortanfang>` : Ausgeben des vollständigen Indexes zu allen Wörtern mit dem Wortanfang `wortanfang` am Terminal
  - `-t=<dateiname>` : Ausgeben der Indizes zu allen Wörtern, die in der Datei `dateiname` vorkommen, am Terminal
  - Die Anfragen verwenden nicht die Eingabedateien, sondern lesen einen zuvor erstellten Index ein! In diesem Fall ist also `outputfile` die einzulesende Datei und es gibt kein `inputfile`!

  `<outputfile>` : Dateiname der Ausgabedatei mit der Indexliste
  `<inputfile>*` : Liste von Eingabedateien mit zu indizierendem Text
- Ausgabe von aussagekräftigen Meldungen bei fehlerhaften Eingaben auf der Kommandozeile
- Verhindern des Überschreibens von Ausgabedateien (leicht)
- Die Ausgabe des vom Programm erzeugten Wortindexes am Terminal und in die Ausgabedatei soll so aufgebaut sein:
  `<wort> (BLANK <dateiindex>)+` , wobei
  `<dateiindex> ::= <dateiname> (BLANK <zeilennummer>)+` ist.
  Dabei beginnen Wörter eine neue Zeile, jeder weitere Dateiname beginnt ebenfalls eine neue Zeile, wird aber mindestens ein Zeichen weit eingerückt.
- Die Ausgabe der Wortliste erfolgt stets lexikografisch sortiert.
- Die Ausgabe der Zeilennummern erfolgt in aufsteigender Reihenfolge.

Anforderungen an die Implementierung

- Als Kodierung der Zeichen in den Textdateien und im Index wird *ISO Latin1* verwendet.
- Es dürfen nur Standardbibliotheken von Haskell und GHC für die Implementierung verwendet werden.

- Es wird ein Makefile bereitgestellt für
  - das Erzeugen des ausführbaren Programms
  - das Löschen aller aus den Programmquellen erzeugten Daten
- Zum Testen des Programms stehen Testdateien in der Datei `Testdaten.zip` auf der Seite zur Lehrveranstaltung zur Verfügung.

Benotung:
Um eine sehr gute Note erreichen zu können, müssen Sie außer den *Muß-Kriterien* auch alle *Soll-Kriterien* erfüllen. Die Erfüllung einzelner *Soll-Kriterien* führt zu einer schrittweisen Verbesserung der Ausgangsnote *drei* für die Bewertung.
Lösungen, die sich nicht an die unter *Anforderungen an die Implementierung* genannten Regeln halten, werden mit **null** Punkten bzw. der Note 5 (**ungenügend**) bewertet!
Beachten Sie bitte dazu auch die auf der Seite zur Lehrveranstaltung veröffentlichte Notenskala!

Als Lösung sind abzugeben:

- ein **Ausdruck** einer schriftlichen Beschreibung der Programmstruktur
- ein **Ausdruck** des kommentierten Programms (z.B. erstellt mittels `pp`)
- ein **Ausdruck** der Indexinformation zu den Texten `Indextest0.txt` und `Indextest1.txt`

Die Abgabe der Lösung erfolgt durch jede Gruppe von Studierenden (maximal 2 Personen pro Gruppe) persönlich im Rahmen der entsprechenden oben genannten Übungsstunde an den Dozenten. Bei der Abgabe der Lösung an einem Rechner im Übungslabor muß das unter Linux funktionsfähige Programm übersetzt und vorgeführt werden. Beide Studierende einer Gruppe müssen alle Fragen des Dozenten zum Programm beantworten können.

Bewertungskriterien:
Bewertet werden neben der Vorführung mit Erläuterung (siehe oben):

- die korrekte Funktion des Programms
- die funktionale Struktur des Programms
- die Robustheit des Programms
- die Lesbarkeit des Programmtextes
- die Vollständigkeit und Verständlichkeit der Programmdokumentation
- die Beschreibung der Programmstruktur
- die Form der schriftlichen Dokumente
- die Extras

A.2. Index aus Testdaten

Index aus indextest0.txt
auch Testdaten/Indextest0.txt 5
Dies Testdaten/Indextest0.txt 1 5
eine Testdaten/Indextest0.txt 1
es Testdaten/Indextest0.txt 7
geht Testdaten/Indextest0.txt 6
ist Testdaten/Indextest0.txt 1
noch Testdaten/Indextest0.txt 5
So Testdaten/Indextest0.txt 6
Testdatei Testdaten/Indextest0.txt 5
zweite Testdaten/Indextest0.txt 2

Index aus indextest1.txt
Alles Testdaten/Indextest1.txt 5
auch Testdaten/Indextest1.txt 4
Dies Testdaten/Indextest1.txt 1 4
eine Testdaten/Indextest1.txt 1
erste Testdaten/Indextest1.txt 2
ist Testdaten/Indextest1.txt 1
noch Testdaten/Indextest1.txt 4
prima Testdaten/Indextest1.txt 5
Testdatei Testdaten/Indextest1.txt 4

A.3. Quellcode

Listing A.1: „main.hs“

```
1  import System.Environment
2  import Data.List
3  import System.Directory
4  import System.IO
5  import System.IO.Unsafe
6
7  import Index
8  import Parser
9  import Cmdlineargs
10
11 type Text = String
12 type Wort = String
13 type File = String
14
15 -- get a list of text and a list of files and make a list of pairs
16 makePair :: [Text] -> [File] -> [(Text,File)]
17 makePair text files = zip text files
18
19 -- function for argument -c
20 printWordNumber :: [(Wort, [(File, [Int])])] -> IO ()
21 printWordNumber index = do
22     putStr "Anzahl der WÃ¶rter: "
23     print (length (map fst index))
24
25 -- function for argument -p
26 printIndex :: [(Wort, [(File, [Int])])] -> IO ()
27 printIndex index = putStr (printAsString index)
28
29 -- function for argument -i
30 createIndex :: [(Text,File)] -> [(Wort, [(File, [Int])])]
31 createIndex content = index content
32
33 -- function for argument -q
34 printIndexForWord :: Wort -> [(Wort, [(File, [Int])])] -> IO ()
35 printIndexForWord word index
36     | length elements > 0 = putStr (printAsString elements)
37     | otherwise = putStrLn ("no elements for word: '" ++ word ++ "' in
          list!")
38     where elements = (filter (\ a -> (fst a) == word) index)
39
40
41 -- function for argument -t
42 printIndexForFile :: File -> [(Wort, [(File, [Int])])] -> IO ()
43 printIndexForFile file index
44     | length elements > 0 = putStr (printAsString elements)
45     | otherwise = putStrLn ("no elements for file: '" ++ file ++ "' in
          list!")
46     where elements = (filter (\ a -> elem file (map fst (snd a)) ) index)
47
48 -- function for argument -s
49 printIndexForWordPartial :: String -> [(Wort, [(File, [Int])])] -> IO ()
50 printIndexForWordPartial partialWord list
51     | length elements > 0 = putStr (printAsString elements)
52     | otherwise = putStrLn ("no elements for string: '" ++ partialWord ++
          "' in list!")
53     where elements = (filter (\ a -> isPartOfWord partialWord (fst a))
          list)
54
55 isPartOfWord :: String -> Wort -> Bool
```

```
56 isPartOfWord [] [] = True
57 isPartOfWord [] _ = True
58 isPartOfWord _ [] = False
59 isPartOfWord (s:str) (w:word)
60     | s == w = True && isPartOfWord str word
61     | otherwise = False
62
63 -- function to write index in file, test if path does exist
64 writeFile' :: File -> String -> IO ()
65 writeFile' path content = do
66     isFilePresent <- doesFileExist path
67     if not isFilePresent then writeFile path content else putStrLn
           "Ausgabedatei schon vorhanden! Beende Programm."
68
69 createOutFile :: File -> IO ()
70 createOutFile path = do
71     isFilePresent <- doesFileExist path
72     if not isFilePresent then writeFile path "" else return ()
73
74 optionsContain :: [(String,String)] -> String -> Bool
75 optionsContain [] _ = False
76 optionsContain (l:list) str
77     | (fst l) == str = True
78     | otherwise = optionsContain list str
79
80 getValueFromOption :: [(String,String)] -> String -> String
81 getValueFromOption [] _ = []
82 getValueFromOption (l:list) str
83     | (fst l) == str = (snd l)
84     | otherwise = getValueFromOption list str
85
86 main = do
87     -- get arguments
88     args <- getArgs
89     let options = getOption args
90     --print options
91
92     let files = getInputFiles args
93     let out_file = getOutputFile args
94
95     --print files
96
97     --print out_file
98
99     content_list <- mapM readFile files
100    let content = makePair content_list files
101
102    --createOutFile out_file
103    if (optionsContain options "-i" && (length files) == 0)
104        then putStrLn "Achtung: Keine Input Datein angegeben!"
105        else return ()
106
107    let idx = if optionsContain options "-i"
108        then createIndex content
109        else do
110            let content_out = unsafePerformIO (readFile out_file)
111            parse content_out
112
113    if optionsContain options "-p"
114        then do
115            putStrLn (printAsString idx)
116            if optionsContain options "-c" then printWordNumber idx else
                   return ()
117        else return ()
```

```
118
119     if optionsContain options "-q"
120         then do
121             printIndexForWord (getValueFromOption options "-q") idx
122             if optionsContain options "-c" then printWordNumber (filter (\
                    a -> (fst a) == (getValueFromOption options "-q")) idx)
                    else return ()
123         else return ()
124
125     if optionsContain options "-s"
126         then do
127             printIndexForWordPartial (getValueFromOption options "-s") idx
128             if optionsContain options "-c" then printWordNumber (filter (\
                    a -> isPartOfWord (getValueFromOption options "-s") (fst
                    a)) idx) else return ()
129         else return ()
130
131     if optionsContain options "-t"
132         then do
133             printIndexForFile (getValueFromOption options "-t") idx
134             if optionsContain options "-c" then printWordNumber (filter (\
                    a -> elem (getValueFromOption options "-t") (map fst (snd
                    a)) ) idx) else return ()
135         else return ()
136
137     -- list with files
138     --let files =
            ["Testdaten/DasSchloss/K1.iso-latin1.txt","Testdaten/DasSchloss/K2.iso-latin1.txt","
139     --let files = ["Testdaten/Euler.txt"]
140     --let files = ["Testdaten/Indextest0.txt"]
141     -- read several files
142     --content_list <- mapM readFile files
143     --let content = makePair content_list files
144     --let idx = createIndex content
145     -- define output file
146     --let outputFile = "out.txt"
147     -- parse outputfile
148     --content <- readFile outputFile
149     --let idx = parse content
150     --printWordNumber idx
151     --putStr (printAsString idx)
152     --printIndexForWord "mo" idx
153     --printIndexForFile "text1.txt" idx
154     --printIndexForWordPartial "m" idx
155
156     -- write in output file
157     --let outputFile = "out.txt"
158     writeFile' out_file (printAsString idx)
```

Listing A.2: „Cmdlineargs.hs“

```
1  module Cmdlineargs
2  ( getOption, getInputFiles, getOutputFile, isValidOption ) where
3
4  import System.Environment(getArgs)
5  import System.IO
6
7  type Option = String
8  type Value = String
9  type Valid = Bool
10
11 -- this function check the number of given parameters
12 checkArgs :: [String] -> Bool
```

```
13 checkArgs [] = False
14 checkArgs (x:xs) = True
15
16 --check the string for - charakter at the front
17 isValidOption :: String -> Bool
18 isValidOption str
19     | str == "-p" = True
20     | str == "-i" = True
21     | str == "-c" = True
22     | str == "-q" = True
23     | str == "-s" = True
24     | str == "-t" = True
25     | otherwise = False
26
27 getOption :: [String] -> [(Option, Value)]
28 getOption [] = []
29 getOption args = removeEqSign (getOption' (getOnlyOptions args))
30
31 -- separate strings and create a list of (String,String) Pair
32 getOption' :: [String] -> [(Option, Value)]
33 getOption' list = [break (=='=') e | e <- list]
34
35 getOnlyOptions :: [String] -> [String]
36 getOnlyOptions list = filter (\ a -> (head a) == '-') list
37
38 removeEqSign :: [(Option, Value)] -> [(Option, Value)]
39 removeEqSign list = [(fst a, filter (\ a -> a /= '=') (snd a))  | a <-
       list]
40
41 --sortOption :: [String] -> [String]
42 --sortOption list = [if (isOption e) then e else "" | e <-list]
43
44 --isValidOption :: [(Option,Value)] -> [(Option,Value)]
45 --isValidOption list = [if isOption (fst e) then ((fst e), (snd e)) else
       ("","") | e <-list]
46
47 --deleteIfEmpty :: [(Option,Value)] -> [(Option,Value)]
48 --deleteIfEmpty pairs = [ | e <- pairs]
49
50 -- filter (\ a -> (head a) /= '-') ["-p, ....]
51 -- head (filter (\ a -> (head a) /= '-') ["-p, ....]
52
53 getInputFiles :: [String] -> [String]
54 getInputFiles [] = []
55 getInputFiles list =
56     if (getNonOptions list) /= []
57         then tail (getNonOptions list)
58         else []
59
60 getNonOptions :: [String] -> [String]
61 getNonOptions [] = []
62 getNonOptions list = filter (\ a -> (head a) /= '-') list
63
64 getOutputFile :: [String] -> String
65 getOutputFile [] = []
66 getOutputFile list =
67     if (getNonOptions list) /= []
68         then head (getNonOptions list)
69         else []
70
71
72 main = do
73     -- Read given commandline arguments
74     args <- getArgs
```

```
75      print args
76      -- Check args for correctness
77      --let isitaoption = isOption (args !! 0)
78      --if isitaoption then print "Is an option --> Must be added to data
            structure" else print "Is not an option"
79
80     -- let options = sortOption args
81     -- let pairOption = getOption args
82     -- let onlyValidOptions = isValidOption pairOption
83     -- print onlyValidOptions
84      let listOptions = getOnlyOptions args
85      let pairOptions = getOption listOptions
86
87
88
89      print pairOptions
90      let inputFiles = getInputFiles args
91      print inputFiles
92
93      let outputFile = getOutputFile args
94      print outputFile
```

Listing A.3: „Index.hs“

```
1  module Index
2  ( index, printAsString) where
3
4  import Data.List (sortBy)
5  import Data.Char (toLower)
6
7  type Text = String
8  type Zeile = String
9  type Wort = String
10 type File = String
11
12 index :: [(Text,File)] -> [(Wort, [(File, [Int])])]
13 index content = sortMe (removeEmptyEntries (removeDoubleElements
       (mergeFiles (merge' (gather (changeStyleOfWords (ignoreHead (words'
       (addLn (ignoreTail (split content))))))))))
14
15 removeEmptyEntries :: [(Wort, [(File, [Int])])] -> [(Wort, [(File,
       [Int])])]
16 removeEmptyEntries list = filter (\ a -> (fst a) /= "") list
17
18 split :: [(Text,File)] -> [([Zeile],File)]
19 split list = [split' pair |Â pair <- list]
20      --where split' p = (lines (fst p), snd p)
21
22 -- trennt text in zeilen auf
23 split' :: (Text,File) -> ([Zeile],File)
24 split' pair = (lines (fst pair), snd pair)
25
26 ignoreTail :: [([Zeile],File)] -> [([Zeile],File)]
27 ignoreTail list = [(ignoreTail' (fst pair), snd pair) |Â pair <- list]
28
29 ignoreTail' :: [Zeile] -> [Zeile]
30 ignoreTail' [] = []
31 ignoreTail' (z:zs) = ignoreTail'' z:ignoreTail' zs
32
33 ignoreTail'' :: Zeile -> Zeile
34 ignoreTail'' [] = []
35 ignoreTail'' (c:cs)
36      | isTailValid (c:[]) = c:(ignoreTail'' cs)
```

```
37      | otherwise = ' ':ignoreTail'' cs
38
39  -- Test if the tail of a word is valid
40  isTailValid :: String -> Bool
41  isTailValid [] = True
42  isTailValid (c:cs)
43      | isAnyCharValid = (True && isTailValid cs)
44      | otherwise = False
45      where isAnyCharValid = isHeadValid c || c == '-' ||Ã (c >= '0' && c <=
            '9')
46
47
48  addLn :: [([Zeile],File)] -> [([(Zeile,Int)],File)]
49  addLn list = [(addLn' (fst pair), snd pair) |Ã pair <- list]
50
51  -- fuege Zeilennummern hinzu
52  addLn' :: [Zeile] -> [(Zeile,Int)]
53  addLn' z = zip z [1..]
54
55  changeStyleOfWords :: [([(Wort,Int)],File)] -> [(Wort, (File, Int))]
56  changeStyleOfWords [] = []
57  changeStyleOfWords (list:lists) = changeStyleOfWords' (fst list) (snd
        list) ++ changeStyleOfWords lists
58
59  changeStyleOfWords' :: [(Wort,Int)] -> File -> [(Wort, (File, Int))]
60  changeStyleOfWords' pairs file = [changeStyleOfWords'' pair file |Ã pair
        <- pairs]
61
62  changeStyleOfWords'' :: (Wort,Int) -> File -> (Wort, (File, Int))
63  changeStyleOfWords'' pair file = (fst pair, (file, snd pair))
64
65
66  words' :: [([(Zeile,Int)],File)] -> [([(Wort,Int)],File)]
67  words' list = [(words'' (fst pair), snd pair) | pair <- list]
68
69  -- erstelle eine liste mit wÃ¶rtern und deren zeilennummer
70  words'' :: [(Zeile,Int)] -> [(Wort,Int)]
71  words'' [] = []
72  words'' (p:ps) = (zip wort_list (replicate (length wort_list) (snd p))) ++
        words'' ps
73      where wort_list = words (fst p)
74
75  ignoreHead :: [([(Wort,Int)],File)] -> [([(Wort,Int)],File)]
76  ignoreHead list = [(ignoreHead' (fst pair), snd pair) |Ã pair <- list]
77
78  ignoreHead' :: [(Wort,Int)] -> [(Wort,Int)]
79  ignoreHead' [] = []
80  ignoreHead' list = [(ignoreHead'' (fst pair), snd pair) | pair <- list]
81
82  ignoreHead'' :: Wort -> Wort
83  ignoreHead'' [] = []
84  ignoreHead'' (c:cs)
85      | isHeadValid (c) = c:cs
86      | otherwise = ignoreHead'' cs
87
88  -- Test if the head of a word is valid
89  isHeadValid :: Char -> Bool
90  isHeadValid x
91      |Ã x == '_' = True
92      | x >= 'A' && x <= 'Z' = True
93      |Ã x >= 'a' && x <= 'z' = True
94      | isUmlaut x = True
95      | otherwise = False
96
```

```
97  isUmlaut :: Char -> Bool
98  isUmlaut x
99      |Â x == 'Ã¤' || x == 'Ã ' = True
100     |Â x == 'Ã¶' || x == 'Ã ' = True
101     | x == 'Ã¼' ||Â x == 'Ã ' = True
102     | x == 'Ã ' = True
103     | otherwise = False
104
105 gather :: [(Wort, (File, Int))] -> [[(Wort, (File, Int))]]
106 gather list = (gather' list [])
107
108 gather' :: [(Wort, (File, Int))] -> [[(Wort, (File, Int))]] -> [[(Wort,
        (File, Int))]]
109 gather' [] temp = temp
110 gather' list temp
111     |Â isWordInTempList = gather' (tail list) temp
112     | otherwise = gather' (tail list) ((collectSameWords (fst (head list))
            list) : temp)
113     where isWordInTempList = (elem (fst (head list)) (map fst (map head
            temp)))
114
115 collectSameWords :: Wort -> [(Wort, (File, Int))] -> [(Wort, (File, Int))]
116 collectSameWords word list = filter (\ e -> (word) == (fst e)) list
117
118 merge' :: [[(Wort, (File, Int))]] -> [(Wort, [(File, Int)])]
119 merge' list = [merge'' sub_list |Â sub_list <- list]
120
121 merge'' :: [(Wort, (File, Int))] -> (Wort, [(File, Int)])
122 merge'' list = (fst (head list), (map snd list))
123
124 mergeFiles :: [(Wort, [(File, Int)])]  -> [(Wort, [(File, [Int])])]
125 mergeFiles [] = []
126 mergeFiles list = mergeFiles' (head list) : mergeFiles (tail list)
127
128 mergeFiles' :: (Wort, [(File, Int)]) -> (Wort, [(File, [Int])])
129 mergeFiles' pair = (fst pair, mergeFiles'' (snd pair) [])
130
131 mergeFiles'' :: [(File, Int)] -> [(File, [Int])] -> [(File, [Int])]
132 mergeFiles'' [] temp = temp
133 mergeFiles'' pairs temp
134     | isFileInTempList = mergeFiles'' (tail pairs) temp
135     | otherwise = mergeFiles'' (tail pairs) ( (fst (head pairs),
            collectSameFiles (fst (head pairs)) (pairs) ) : temp)
136     where isFileInTempList = (elem (fst (head pairs)) (map fst temp))
137
138 collectSameFiles :: File -> [(File, Int)] -> [Int]
139 collectSameFiles file list = map snd (filter (\ e -> (file) == (fst e))
        list)
140
141 removeDoubleElements :: [(Wort, [(File, [Int])])] -> [(Wort, [(File,
        [Int])])]
142 removeDoubleElements list = [(fst pair, removeDoubleElements' (snd pair))
        |Â pair <- list]
143
144 removeDoubleElements' :: [(File, [Int])] -> [(File, [Int])]
145 removeDoubleElements' list = [removeDoubleElements'' pair | pair <- list]
146
147 removeDoubleElements'' :: (File, [Int]) -> (File, [Int])
148 removeDoubleElements'' pair = (fst pair, (removeDoubleElements'''( reverse
        (snd pair)) []) )
149
150 removeDoubleElements''' :: [Int] -> [Int] -> [Int]
151 removeDoubleElements''' [] temp = temp
152 removeDoubleElements''' (l:list) temp
```

```
153     | (elem l temp) = removeDoubleElements''' list temp
154     | otherwise = removeDoubleElements''' list (l:temp)
155
156 sortMe :: [(Wort, [(File, [Int])])] -> [(Wort, [(File, [Int])])]
157 sortMe list = sortBy (\ x y -> compareMe (fst x) (fst y)) list
158
159 compareMe :: Wort -> Wort -> Ordering
160 compareMe [] [] = EQ
161 compareMe [] _ = LT
162 compareMe _ [] = GT
163 compareMe w1 w2
164     | ord_case_insensitive == EQ = compareMe (tail w1) (tail w2)
165         --if ord_case_sensitive == EQ
166             --then compareMe (tail w1) (tail w2)
167             --else ord_case_sensitive
168     | otherwise = ord_case_insensitive
169     where ord_case_insensitive
170             | (isUmlaut (head w1) && isUmlaut (head w2)) = compareMe
                    (replaced_w1 ++ (tail w1)) (replaced_w2 ++ (tail w2))
171             | isUmlaut (head w1) = compareMe (replaced_w1 ++ (tail w1)) w2
172             | isUmlaut (head w2) = compareMe w1 (replaced_w2 ++ (tail w2))
173             |Â otherwise = compareInsensitive (head w1) (head w2)
174             where replaced_w1 = replaceUmlautInWort (head w1)
175                   replaced_w2 = replaceUmlautInWort (head w2)
176           {-ord_case_sensitive
177             | isUmlaut (head w1) && isUmlaut (head w2) = compareMe
                    (replaced_w1 ++ (tail w1)) (replaced_w2 ++ (tail w2))
178             | isUmlaut (head w1) = compareMe (replaced_w1 ++ (tail w1)) w2
179             | isUmlaut (head w2) = compareMe w1 (replaced_w2 ++ (tail w2))
180             | otherwise = compareSensitive (head w1) (head w2)
181             where replaced_w1 = replaceUmlautInWort (head w1)
182                   replaced_w2 = replaceUmlautInWort (head w2)-}
183
184 -- case insensitive compare
185 compareInsensitive :: Char -> Char -> Ordering
186 compareInsensitive c1 c2 = compare (toLower c1) (toLower c2)
187
188 -- case sensitive compare
189 compareSensitive :: Char -> Char -> Ordering
190 compareSensitive c1 c2 = compare c1 c2
191
192 containUmlaute :: Wort -> Bool
193 containUmlaute [] = False
194 containUmlaute (w:wort)
195     | isUmlaut w = True
196     | otherwise = containUmlaute wort
197
198 replaceUmlautInWort :: Char -> Wort
199 replaceUmlautInWort c
200     | c == 'Ã¤' = "ae"
201     | c == 'Ã ' = "Ae"
202     | c == 'Ã¶' = "oe"
203     |Â c == 'Ã ' = "Oe"
204     | c == 'Ã¼' = "ue"
205     | c == 'Ã ' = "Ue"
206     | c == 'Ã ' = "ss"
207     | otherwise = c:[]
208
209
210 printAsString :: [(Wort, [(File, [Int])])] -> String
211 printAsString [] = ""
212 printAsString (l:list) = ((fst l) ++ (printFileListAsString (snd l)) ++
        (printAsString list))
213
```

```
214 printFileListAsString :: [(File, [Int])] -> String
215 printFileListAsString [] = ""
216 printFileListAsString (list:lists) = (" " ++ (printFileAsString (fst list)
        (snd list)) ++ "\n" ++ (printFileListAsString lists))
217
218 printFileAsString :: File -> [Int] -> String
219 printFileAsString file list = (file ++ (printLineNrAsString list))
220
221 printLineNrAsString :: [Int] -> String
222 printLineNrAsString [] = ""
223 printLineNrAsString (l:list) = (" ") ++  (show l) ++ (printLineNrAsString
        list)
```

Listing A.4: „Parser.hs“

```
 1 module Parser
 2 ( parse ) where
 3
 4 import Data.Char(isSpace, isNumber)
 5
 6 type Text = String
 7 type Zeile = String
 8 type Wort = String
 9 type File = String
10
11 parse :: String -> [(Wort, [(File, [Int])])]
12 parse content = parse' (lines content)
13
14 parse' :: [String] -> [(Wort, [(File, [Int])])]
15 parse' [] = []
16 parse' (l:list)
17     | isWordLine l = (fst word_pair, (snd word_pair) ++ (map parseFileLine
           (nextFilesLine list)) ) : parse' list
18     | otherwise = parse' list
19     where
20         word_pair = parseWordLine l
21
22 isWordLine :: String -> Bool
23 isWordLine [] = False
24 isWordLine (c:_)
25     | not (isSpace c)  = True
26     | otherwise = False
27
28 nextFilesLine :: [String] -> [String]
29 nextFilesLine [] = []
30 nextFilesLine (l:list)
31     | isFileLine l = l : (nextFilesLine list)
32     | otherwise = []
33
34 isFileLine :: String -> Bool
35 isFileLine [] = False
36 isFileLine (c:_)
37     | isSpace c = True
38     | otherwise = False
39
40 parseWordLine :: Zeile -> (Wort, [(File, [Int])])
41 parseWordLine z = (fst wort_pair, [(fst file_pair, zeilen)])
42     where
43         wort_pair = parseWort z
44         file_pair = parseFile (snd wort_pair)
45         zeilen = parseLines (snd file_pair)
46
47 parseFileLine :: Zeile -> (File, [Int])
```

```
48 parseFileLine z = (fst file_pair, zeilen)
49    where
50        file_pair = parseFile z
51        zeilen = parseLines (snd file_pair)
52
53 parseWort :: Zeile -> (Wort,Zeile)
54 parseWort s = (head (words s), unwords (tail (words s)))
55
56 parseFile :: Zeile -> (File, Zeile)
57 parseFile f = (head (words f), unwords (tail (words f)))
58
59 parseLines :: Zeile -> [Int]
60 parseLines l = [read w :: Int |Ã w <- (words l)]
```

A.4. Makefile

Listing A.5: „Makefile“

```
1  # This is a makefile created by Christian Bunk
2  # Last update 03.07.2011
3
4  # Files to be compiled
5  OBJECTS = $(SRC)/Main.hs \
6  $(SRC)/Index.hs \
7
8  # program name
9  NAME = index
10
11 # source directory
12 SRC = ./src
13
14 install:
15     ghc --make $(SRC)/Main.hs $(SRC)/Index.hs $(SRC)/Parser.hs
           $(SRC)/Cmdlineargs.hs -o index
16
17 clean:  FORCE
18     rm -f $(SRC)/*.o $(SRC)/*.hi
19
20 FORCE:
```