Informatik–Seminar im Sommersemester 2001

Thema:	CGI–Programmierung mit **Haskell**
Dozent:	Prof. Dr. Uwe Schmidt
Student:	Markus Schwarz
Matrikelnummer:	wi7715
Datum:	09.05.2001

Inhaltsverzeichnis

1 Einleitung

1.1 Ausblick

Das Ziel dieser Ausarbeitung ist es, die Möglichkeiten von **Haskell** für die CGI Programmierung anhand einfacher Beispiele darzustellen. Grundkenntnisse der funktionalen Programmierung und speziell **Haskell** sind hilfreich, aber nicht erforderlich, da alle benutzten Sprachkonstrukte im Abschnitt 2 erläutert werden.

1.2 Was ist funktionale Programmierung?

Funktionale Programmiersprachen bestehen ausschließlich aus Funktionen. Ein funktionales Programm selbst ist eine Funktion, die Eingaben entgegen nimmt und ihre Ausgabe als Resultat liefert. In den meisten Fällen wird diese Hauptfunktion nicht in der Lage sein, das Ergebnis mit Hilfe elementarer Sprachelemente direkt zu berechnen, sondern wird sich anderer Funktionen bedienen und deren Ergebnisse miteinander kombinieren. Während es sich bei imperativen Programmiersprachen um eine Reihe von Kommandos handelt, die in einer bestimmten Reihenfolge ausgewertet werden, spielt die Reihenfolge in der Welt der funktionalen Sprachen normalerweise keine Rolle. Vielmehr handelt es sich um Ausdrücke (Formeln), die je nach Bedarf ausgewertet werden. Hierbei ist zu beachten, daß es in funktionalen Sprachen keine Variablen mit wechselnden Werten gibt, sondern nur Funktionen, die einen Wert liefern. Dabei ist eine Funktion kein Verweis auf eine Speicheradresse sondern kann als eine Konstante betrachtet werden. Jeder Funktionsaufruf mit den selben Parametern führt auch immer zum selben Resultat. Per Definition sind Seiteneffekte somit ausgeschlossen.

Die modulare Softwareentwicklung wird heute als *good practice* angesehen. Da funktionale Sprachen Modularität besonders gut unterstützen, sind die Programme weitaus kürzer, besser wartbar und weniger anfällig gegen Fehler als in imperativen Sprachen geschriebene. Zwei Aspekte, die die Modularität fördern sind *Higher Order Functions* (siehe 2.4.1) und *Lazy Evaluation* (siehe 2.4.2). Ein weiteres wichtiges Element funktionaler Sprachen ist die Rekursion. Da Schleifen in den meisten Fällen nicht ohne Zuweisungen auskommen, gibt es sie in funktionalen Sprachen nicht. Aus diesem Grund werden Wiederholungen durch Rekursion abgebildet.

2 Einführung in Haskell

Für all die jenigen, die bislang noch nicht in einer funktionalen Sprache programmiert haben, möchte ich an dieser Stelle eine kleine Einführung in **Haskell** liefern. Es bleibt zu bemerken, daß diese Ausarbeitung allerdings nur einen geringen Teil der Möglichkeiten dieser mächtigen Sprache abdecken kann.

2.1 Funktionen

Eine Funktion ist eine Art *Black Box*, die aus belibig vielen Eingabewerten eine Ausgabe berechnet. In den meisten Fällen wird diese Funktion ihre Arbeit wieder an weitere Funktionen delegieren und nur deren Ausgabe geeignet verknüpfen. Eine Funktion ist eine Definition, die einen Namen (Bezeichner) mit einem Wert eines bestimmten Types verbindet. Eine der einfachsten denkbaren Deklarationen einer Funktion ist:

```
antwort :: Int
antwort = 42
```

Hierbei wird der Bezeichner `antwort` vom Typ `Int` mit der Konstante `42` verknüpft.

Funktionsdeklaration

Die Deklaration (in objektorientierten Sprachen auch als Signatur bezeichnet) ist unterteilt in einen Funktionsbezeichner, der mit einem Kleinbuchstaben beginnen muß, einer optionalen Menge von formalen Parametern und einem zwingend vorgeschriebenen Rückgabetyp (siehe Kapitel 2.2). Funktionsbezeichner und Parameter werden duch `::` voneinander getrennt. Die einzelnen Eingabeparameter und der Rückgabetyp werden durch `->` getrennt.

Beispielhaft läßt sich dies an der Funktionsdeklaration an der Funktion

```
fac :: Int -> Int
```

erläutern. Aufgabe von `fac` ist es, die Fakultät einer beliebigen ganzen Zahl zu berechnen und das Ergebnis (ebenfalls eine ganze Zahl) zurückzuliefern.

Funktionsrumpf

Der Rumpf einer **Haskell** Funktion ist typischerweise sehr kurz, da entweder kleine elementare Operationen ausgeführt werden oder andere Funktionen aufgerufen werden.

Typischerweise besteht der Rumpf aus dem Funktionsbezeichner, den Namen der formalen Parameter, einem Gleichheitszeichen und der Berechnungsvorschrift. Die Funktion `inc` berechnet den Nachfolger einer ganzen Zahl. Sie kann wie folgt definiert werden.

```
inc :: Int -> Int
inc n = n + 1
```

Die vollständige Definition der oben beschriebenen Funktion `fac` zeigt ein mächtiges Hilfsmittel von **Haskell** . Die Rede ist von *Pattern Matching*. Beim Pattern Matching werden die Parameter der Funktion ausgewertet, um in Abhängigkeit

dieser Parameter alternative Berechnungsvorschriften aufzurufen. Wie bereits erwähnt ist die Rekursion wegen fehlender Schleifen ein integraler Bestandteil der funktionalen Programmierung. Am Beispiel der Funktion `fac` wird deutlich, warum Pattern Matching besonders für die Rekursion von großer Bedeutung ist.

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n - 1)
```

Die Berechnung der Fakultät erfolgt durch die Multiplikation der angegebenen Zahl n mit der Fakultät von $n - 1$. Ist die Zahl $n = 0$, wird durch Pattern Matching definitionsgemäß 1 zurückgegeben. Für alle $n \neq 0$ wird die rekursive Berechnungsvorschrift aufgerufen.

2.2 Typen

Standard Datentypen

Es gibt eine Reihe von vordefinierten Datentype wie z.B. `Char`, `Int`, `Bool`, `Float` und `String` wobei `String` intern als eine Liste von `Char` representiert wird (2.3).

Benutzerdefinierte Datentypen

Es gibt in **Haskell** diverse Möglichkeiten, eigene Datentypen zu definieren. Im folgenden werde ich lediglich die algebraischen Datentypen (Aufzählungstyp und Produkttyp) und die Synonyme betrachten. Auf Grund der Komplexität kann auf Klassen hier nicht näher eingegangen werden.

Mit Hilfe von Synonymen lassen sich Programme lesbarer machen und zusammengesetzte Datentypen erstellen. Beispielsweise ist wie bereits erwähnt ein `String` eine Liste von `Char` und eine Person kann als Tupel aus Name und Alter aufgefaßt werden. Die Deklaration eines Synonyms erfolt durch das Schlüsselwort `type`, dem Typnamen, einem Gleichheitszeichen und der Definition. Alle Typnamen in **Haskell** beginnen mit einem Großbuchstaben.

```
type String = [Char]
type Person = (String, Int)
```

Alle algebraischen Datentypen lassen sich durch das Schlüsselwort `data`, dem Typnamen, einem Gleichheitszeichen und der Definition erzeugen.

Ein Auzählungstyp ist ein Typ mit festem Wertebereich wie zum Beispiel die Monate eines Jahres.

```
data Monat = Jan | Feb | Mar | Apr | Mai | Jun | Jul |
             Aug | Sep | Okt | Nov | Dez
```

Ein Produkttyp kann mit einem Pascal Record verglichen werden. Beliebig viele Daten unterschiedlichen Typs können zu einem Produkttyp zusammengefaßt werden. Diese Typen lassen sich ausschließlich durch definierte Konstruktoren erzeugen. Ein Beispiel für einen Produkttyp ist:

```
type Name = String
type Alter = Int
data Person = Teilnehmer Name Alter
```

Ein Datum des Typs `Person` läßt sich ausschließlich durch den Konstruktoraufruf `Teilnehmer "Markus" 25` erzeugen. Mit Hilfe von Produkttypen lassen sich auch Bäume und andere rekursive Datenstrukturen wie zum Beispiel

```
data Expr = Lit Int | Add Expr Expr | Sub Expr Expr
```

abbilden. Es gibt drei mögliche Ausprägungen des Typs `Expr`. Entweder handelt es sich um ein Blatt (einer Zahl) oder einen Knoten (entweder `Add` oder `Sub`). Erzeugt wird ein Rechnbaum für den Ausdruck `(3-1)+2` durch:

```
Add (Sub (Lit 3) (Lit 1)) (Lit 2)
```

Sowohl die vordefinierten Typen als auch benutzerdefinierte Typen und Synonyme sind gültige formale Parameter und Rückgabewerte von Funktionen.

2.3 Listen

Ein großer Vorteil von **Haskell** gegenüber anderen Sprachen (in diesem Fall besonders gegenüber `C`) liegt im Umgang mit Listen, denn sie sind in **Haskell** von zentraler Bedeutung. ein Listentyp wird definiert durch das Einschließen eines Typs in eckige Klammern. Alle Listenoperationen arbeiten unabhängig vom Typ der in der Liste enthaltenen Werte.

```
length [1, 2, 3] → 3
length ['a', 'b', 'c'] → 3
length [Jan, Feb, Mar] → 3
```

Wie oben verdeutlicht, arbeitet die Funktion `length` auf jeder Art von Liste. Man bezeichnet sie deshalb auch auls polymorph. Die Funktionsdeklaration sieht wie folgt aus: `length [a] -> Int`. Im Sprachumfang von **Haskell** sind noch eine Reihe weiterer Listenoperationen vorgesehen. Kurz erwähnt seien an dieser Stelle die später benutzen Funktionen `:` (ein Element an den Anfang einer Liste hängen), `++` (zwei Listen gleichen Typs zu einer vereinen) und `!!` (indizierter Zugriff auf ein Element einer Liste.

Ausdruck	Ergebnis
`'W' : "ort"`	`"Wort"`
`"Wo" ++ "rt"`	`"Wort"`
`"Wort" !! 2`	`'r'`

2.4 Besonderheiten

2.4.1 Higher Order Functions

Haskell bietet beim Umgang mit Funktionen eine Reihe von Möglichkeiten, die beim Programmieren in imperativen oder objektorientierten Sprachen fehlen oder nur umständlich abzubilden sind. Hierzu zählt das Konzept der *Higher-order functions*. Funktionen sind höherer Ordnung wenn sie eine Funktion als Parameter oder als Rückgabetyp (oder beides) haben. Dies ermöglicht eine ganze Reihe von Anwendungen. Im folgenden werde ich mich auf die Funktion `map` berschränken.

Die Funktion `map :: (a -> b) -> [a] -> [b]` wendet eine ihr gegebene Funktion (mit der Signatur `f :: a -> b`) auf jedes Element der Liste des Typs `a` an und gibt als Erbegnis eine Liste des Typs `b` zurück.

Zur Veranschaulichung hier ein einfaches Beispiel. Angenommen es existiert eine Funktion `show`,

```
show :: Int -> String
```

die eine Zahl in einen String umzuwandelt. Nun habe ich eine Liste von Zahlen

```
[1, 2, 3, 4, 42]
```

und möchte diese in eine Liste von entsprechenden Strings

```
["1", "2", "3", "4", "42"]
```

transorfmieren. In **Haskell** läßt sich diese Aufgabe kurz durch

```
map show [1, 2, 3, 4, 42]
```

lösen. Die Funktion `show` wird auf jedes Element der Liste angewendet und das Resultat in einer Ergebnisliste gespeichert.

2.4.2 Lazy Evaluation

Lazy Evaluation bezeichnet die Fähigkeit zu erkennen, ob ein Ausdruck ausgewertet werden muß, um das Ergebnis zu berechnen oder ob darauf verzichtet werden kann. Beispielsweise ist jeder weitere Parameter einer boolschen OR Verknüpfung gleichgültig, wenn irgend ein anderer wahr ist. In Verbindung mit dem `.`-Operator lassen sich ohne auch nur eine Zeile eigenen Code Pipes, wie man sie aus der Unix Welt kennt, nachbauen.

Die Ausgabe einer Funktion `producer` wird als Eingabe einer Funktion `consument` verwendet, wobei zu beachten ist, daß Funktion `producer` nur so lange ausgeführt wird, wie die Funktion `consument` Eingaben benötigt. Dies sei wiederum an einem kurzen praktischen Beispiel verdeutlicht.

```
queens :: Int -> [[Int]]
...
firstQueen :: Int -> [Int]
firstQueen n = (head.queens) n
```

`queens` berechnet alle möglichen Lösungen für das Damenproblem[1] für n Damen. Die Liste von Ergebnissen wird zur Eingabe für die Listenfunktion `head :: [a] -> a`. Diese Funktion liefert das erste Element einer Liste. Da diese Funktion ausgewertet werden kann bevor alle Ergebnisse aus `queens` vorliegen, bricht auch die Ausführung von `queens` vorzeitig ab. Gerade bei großen n spart diese Art der Auswertung erheblich Rechenzeit, da nur die erste Lösung gefunden werden muß.

2.4.3 Lambda Notation

Eine weitere Möglichkeit von Funktionsdefinitionen sind Funktionen in Lambda-Notation. Dabei handelt es sich um anonyme Funktionen, die an der Stelle definiert werden an der sie verwendet werden. Sie werden definiert mit einem `\` gefolgt von der Parameterliste, einem `->` und der eigentlichen Definition. Hierzu ein Beispiel:

```
map (\n -> n + 1) [1, 2, 3, 4]
```

Wie wir in Abschnitt 2.4.1 gesehen haben, wendet die Funktion `map` eine ihr zu übergebende Funktion auf eine ihr zusätzlich zu übergebende Liste an. In diesem Beispiel wird allerdings kein Funktionsbezeichner übergeben sondern die Funktionsdefinition selbst. Die in Lambda Notation geschriebene Funktion `\n -> n + 1` hat einen Parameter, zu dem sie `1` addiert.

3 Die CGI-Library

Wie im ersten Teil gezeigt, lassen sich einige Probleme in **Haskell** sehr viel eleganter lösen als in imperativen Programmiersprachen. Da liegt der Schluß nahe, diese Vorzüge auch bei der dynmischen Erzeugung von Webseiten zu nutzen. Ein erstes *Hello World* Beispiel läßt sich leicht erzeugen.

```
(1)   #! /usr/bin/runhugs
```

[1] Wie lassen sich n Damen auf einem $n * n$ großen Schachbrett derart anordnen, daß sie sich gegenseitig nicht bedrohen?

```
(2)   main :: IO()
(3)   main = putStr ("Content-type: text/html\n\n"
        ++ "<html>"
        ++ "<head><title>HelloWorld</title></head>"
        ++ "<body><h1>Hello World</h1></body>"
        ++ "</html>")
```

Das Skript wird vom Kommandozeilen-Interpreter `runhugs` ausgeführt und seine Ausgabe als HTTP-Response an den Client zurückgeschickt.

Diese Art der Generierung von HTML Seiten ist sehr mühsam und auch fehleranfällig. Aus diesem Grund haben mehrere Autoren HTML-Bibliotheken entwickelt, die dem Programmierer Funktionen bieten mit deren Hilfe er sehr einfach valide Dokumente erzeugen kann. Die im folgenden näher betrachtete cgi-Bibliothek von Erik Meijer[2] bietet dem Benutzer darüber hinaus eine Schnittstelle mit deren Hilfe er von der gesamten Low-Level Kommunikation via HTTP abstrahieren kann. Dies hat den Vorteil, daß sehr viel mehr Zeit in die Lösung des eigentlichen Problems gesteckt werden kann.

3.1 Aufbau und Arbeitsweise

Um die vielen Details des HTTP für den Entwickler der Skripte transparent zu halten, besteht die Bibliothek aus zwei Komponenten (siehe Abbildung 1). Einem Wrapper und einem Worker. Die Aufgaben, die bei jedem cgi-Skript gleich sind, werden vom Wrapper übernommen. Die eigentlich Programmlogik wird im Worker gekapselt. Durch diesen Ansatz ist es möglich, kürzere und besser wartbare Programme zu schreiben.

Der Ablauf im Wrapper stellt sich in etwa wie folgt dar.

```
      wrapper :: (Request -> IO Response) -> IO ()
(1)   wrapper = \worker ->
        do {
(2)       request <- getRequest
(3)       ; response <- worker request
(4)       ; putResponse response
        }
      'catch'
(5)     (\ioerror -> do {putResponse(status ioerror)})
```

Die Definition des Wrapper ist ein Beispiel für die Nutzung von Higher Order Functions (siehe 2.4.1). Der Funktion `wrapper` wird eine Funktion mit der Signatur `(Request -> IO Response)` übergeben, die in (1) als worker benannt und in (3) aufgerufen wird. Vor dem Aufruf des Workers wird der low-level Request

[2] http://www.cse.ogi.edu/erik/Personal/cgi.htm

Abbildung 1: Architektur der CGI Bibliothek

in einen abstrakten Request vom Typ `Request` umgeformt. Dieser Request wird dem Worker als Parameter mitgegeben. Das Ergebnis des Workers (ein Wert vom Typ `Response`) wird in (4) in einen low-level Response umgewandelt und dem Client zurückgeschickt. Im Falle eines Fehlers wird der Fehlercode automatisch an den Client übermittelt.

Ich verzichte in dieser Ausarbeitung bewußt auf die Details zur Implementierung dieser Umformungen, da sie – wie bereits mehrfach erwähnt – für die Benutzung der Bibliothek keine Rolle spielen. Der interessierte Leser sei auf die exzellente Dokumentation von Meijer verwiesen.

3.2 Modellieren von HTML

Ein Dokument, das in der *Hypertext Markup Language (HTML)* beschrieben ist besteht aus einer Reihe von in einander geschachtelten Elementen wie beispielsweise Überschriften, Tabellen, Aufzählungen, Verknüpfungen und Bilder. In den meisten Fällen werden diese Elemente durch ein öffendes und eine schließen-

des Tag `<tag>text</tag>` beschrieben. Häufig lassen sich die Elemente durch Einfügen optionaler Attribute der Form `name="wert"` parametrisieren. Die hier vorgestellte Bibliothek definiert einen Datentyp HTML als:

```
data HTML = Text String | Element Tag [(Name, Value)] [HTML]
```

Die Typen Tag, Name und Value sind Synonyme für den Datentyp String (vergleiche Abschnitt 2.2). Somit ist ein Wert des Typs `HTML` entweder einfacher Text oder ein Element. Ein Element besteht aus einem Tagnamen, beliebig vielen Attributen und beliebig vielen weiteren Elementen. Um diese Elemente nicht per Hand erzeugen zu müssen stehen selbstverständlich Funktionen bereit, die diese Aufgabe übernehmen. Eine Übersicht über die wichtigsten Funktionen zeigt Abblildung 2. Unter Zuhilfenahme dieser Funktionen lassen sich sehr einfach selbst komplexe Seiten erzeugen.

```
page :: String -> [(Name,Value)] -> [HTML] -> HTML

h1 :: String -> HTML
h :: Int ->String -> HTML
p :: [HTML] -> HTML
hr :: HTML

href :: URL -> [HTML] -> HTML
img :: String -> URL -> HTML

ol :: [[HTML]] -> HTML
ul :: [[HTML]] -> HTML
dl :: [(String,[HTML])] -> HTML

 table :: String -> [String] -> [[ [HTML] ]] -> HTML
```

Abbildung 2: Einige wichtige Funktionen zur HTML-Modellierung

3.3 Eine einfache Seite

Kommen wir noch einmal auf das *Hello World* Beispiel zurück. Der in Abschnitt 3 recht umständlich erzeugte Response kann mit Hilfe der von Erik Meijer erstellten cgi-Bibliothek wie folgt geschrieben werden.

```
import CGI

hello :: HTML
```

```
hello = page "Hello World" [("bgcolor", "#000000")] [h1 "Hello World"]

main :: IO ()
main = wrapper (\env -> do
  return (Content(hello))
  )
```

Bei diesem zugegebenermaßen sehr einfachen Beispiel ist der Schreibaufwand nicht viel geringer als bei der ersten Version, aber das Programm gewinnt stark an Lesbarkeit und Wartbarkeit. Je umfangreicher und komplexer die zu erstellende Seite ist, desto höher sind diese Vorteile einzuschätzen.

3.4 Lesen von Umgebungsvariablen

Wie bereits erwähnt ist es für den Worker (das eigentliche Skript) vollständig transparent, wie die Umgebungsvariablen gelesen werden. Dementsprechend einfach ist es für den Worker, diese zu extrahieren. Beim Aufruf des Wrappers werden die Umgebungsvariablen und die dem cgi-Skript übergebenen Variablen direkt als Liste von Schlüssel-Wert Paaren übergeben. Das nachstehende Beispiel listet alle übergebenen Umgebungsvariablen auf eine HTML Seite auf. Die Liste enthält auch alle die durch die URL zusätzlich übergebenen Parameter. Selbstverständlich läßt sich mit Hilfe der Funktion `lookup "HOST" env` auch indiziert auf eine spezielle Variable zugreifen.

```
seite :: [(Name, Value)] -> HTML
seite env = page "Umgebungsvariablen" []
  [
    h1 "Umgebungsvariablen",
    dl (map (\(dt,dd) -> (dt, [prose dd])) env)
  ]

main :: IO ()
main = wrapper(\env -> do return (Content(seite env)))
```

4 Fallbeispiel Damenproblem

Wie schon an anderer Stelle besprochen handelt es sich beim sogenannten *Damenproblem* um die Frage wie sich n Damen auf einem $n * n$ großen Schachbrett anordnen lassen, ohne daß sie sich gegenseitig bedrohen. Das bedeutet, daß pro Zeile und Spalte genau eine Dame stehen muß und daß nicht zwei Damen auf einer Diagonalen stehen dürfen. Das Hauptaugenmerk liegt in diesem Fallbeispiel nicht auf dem Algorithmus, sondern vielmehr in der Darstellung der Lösung.

Die Aufgabe besteht aus zwei Teilen. Zum einen soll der Benutzer über ein Formular folgende Parameter einstellen können:

- Die Anzahl der Damen(Freitext)
- Die wievielte ermittelte Lösung soll angezeit werden(Freitext)
- Umrandung der Tabelle? (Ja/Nein)(Radiogroup)
- Welches Zeichen soll für die Damen verwendet werden(Auswahlliste)

4.1 Das Formular

Für die Eingabe der oben beschrieben Parameter wird ein Formular verwendet, bei dem alle einzutragenen Felder untereinander aufgeführt sind. Die Entscheidung darüber, ab das Formular oder die Lösung angezeigt wird, wird anhand des Parameters Anzahl gefällt. Ist dieser Wert nicht gesetzt, erscheint das Formular. Anderenfalls wird eine Lösung berechnet.

Wie einfach es ist Fomularelemente zu erzeugen, läßt sich am besten anhand zweier Beispiele belegen. Das Textfeld in dem der Benutzer angeben kann für wie viele Damen er eine Lösung berechnet haben möchte wird durch den Aufruf

```
set [("SIZE", "2"), ("VALUE", "4")] (textfield "anzahl")
```

erzeugt. Hierbei handelt es sich um ein Textfeld namens *anzahl* und den durch die Funktion `set` zusätzlich hinzugefügten Attributen `SIZE="2"` und `VALUE="4"`.

Die Listbox für die Auswahl des Zeichens für die Dame läßt sich sehr einfach durch

```
menu "zeichen" ["O", "X", "+", "0"]
```

bewerkstelligen. Der daraus resultierende HTML Quellcode sieht folgendermaßen aus.

```
<SELECT NAME="zeichen" >
  <OPTION> O </OPTION>
  <OPTION> X </OPTION>
  <OPTION> + </OPTION>
  <OPTION> 0 </OPTION>
</SELECT>
```

4.2 Formatierung der Lösung

Nachdem durch Auswertung der Parameter eine Lösung der Form [2,4,1,3] ermittelt wurde, ist es nun an der Zeit, diese in einer Tabelle abzubilden. Die Lösung soll wie folgt interpretiert werden. In Zeile 1 steht die Dame in Spalte 2, in den Zeilen 2, 3 und 4 entsprechend in den Spalten 4, 1 und 3. Die Generierung der Lösungstabelle vollzieht sich in mehreren Schritten. Mit Hilfe von Rekursion bildet die Funktion `werte` eine Liste von Listen von Listen von Elementen des Typs HTML. Dies ist nötig, da eine Tabelle aus Listen von Zeilen besteht, die ihrerseits eine Liste von Zellen sind, die wiederum eine Liste von HTML Elementen sind. Als Parameter bekommt `werte` die Anzahl der Damen (also Zeilen und Spalten), das Zeichen für eine Dame und die Lösung selbst.

```
werte :: Int -> String -> [Int] -> [[[HTML]]]
werte a z (x:[]) = [zeile x a z]
werte a z (x:xs) = [zeile x a z] ++ (werte a z xs)
```

Mit Hilfe von Pattern Matching wird überprüft, ob es noch weitere Lösungszeilen zu formatieren gilt. Ist dies der Fall, wird die aktuelle Zeile formatiert und die Funktion mit dem Rest der Lösungsliste erneut aufgerufen. Das Formatieren einer Zeile erfolgt durch die Funktion `zeile`, die als Parameter die Spalte in der sich die Dame befindet, die Anzahl aller zu erzeugender Spalten und das Zeichen, mit dem die Dame dargestellt werden soll, erhält. Wiederum durch Rekursion ruft sie für jede zu erstellende Spalte die Funktion `zeichen` auf und generiert somit eine Liste von Zellen (die wie erwähnt eine Liste von HTML Elementen ist).

```
zeile :: Int -> Int -> String -> [[HTML]]
zeile m 1 dame = [[zeichen m 1 dame]]
zeile m n dame = (zeile m (n - 1) dame) ++ [[zeichen m n dame]]
```

Die Funktion `zeichen`

```
zeichen m n dame
  | m == n  = prose dame
  | otherwise = prose "."
```

vergleicht lediglich die beiden hereingegebenen Zahlen und liefert bei Gleichheit das gewünschte Zeichen für die Dame, sonst einen `"."`.

5 Fazit

Haskell ist eine sehr mächtige Sprache mit der sich auch Web Anwendungen einfach und schnell erstellen lassen. Durch das Einbinden einer Bibliothek wie der von Erik Meijer, lassen sich Programme schnell und komfortabel entwickeln.

Benutzt man HTML als Beschreibungssprache für Dokumente, ist der Umfang dieser Bibliothek in jedem Fall ausreichend. Versucht man allerdings Layout damit zu betreiben, stößt man schnell an ihre Grenzen.