Praktikum BKSPP

Organisatorisches und Hintergrundwissen

Sommersemester 2012

Dr. David Sabel

Institut für Informatik
Fachbereich Informatik und Mathematik
Goethe-Universität Frankfurt am Main
Postfach 11 19 32
D-60054 Frankfurt am Main
Email: sabel@ki.informatik.uni-frankfurt.de

Stand: 11. April 2012

Inhaltsverzeichnis

1 Allgemeines

1.1 Organisatorisches

1.1.1 Webseite

Die Webseite zum Praktikum ist unter

`http://www.ki.informatik.uni-frankfurt.de/lehre/SS2012/BKSPP-PR/`

zu finden. Neben dieser Anleitung und Aufgabenstellungen sind dort Verweise auf weiteres Material sowie aktuelle Informationen zu finden.

1.1.2 Regelmäßiges Treffen & Rechnerraum

Mittwochs, um 14 c.t. findet in Seminarraum 9 ein regelmäßiges Treffen aller Praktikumsteilnehmer statt. Hier sollen Fragen und Probleme diskutiert werden, die Anwesenheit ist somit i.A. erforderlich. Genaue Termine bzw. ob das regelmäßiges Treffen ausfällt usw. finden sich auf der Webseite zum Praktikum.

Dienstags von 14-18 ist der Rechnerraum 026 im Keller (Robert-Mayer-Str. 11-15) für das Praktikum reserviert. Wer möchte kann zu dieser Zeit die Rechner in diesem Raum nutzen. (Schlüssel bei D. Sabel abholen).

1.1.3 Modulprüfung

Die Anmeldung zur Modulprüfung ist vom 10. April bis zum 30. April über das QIS/LSF-System möglich.

Die Modulprüfung für Bachelorstudierende wird für die korrekte und vollständige Bearbeitung der Aufgaben vergeben.

Es sollen je **4** Teilnehmer für das Praktikum eine Gruppe bilden.

Zur Erfüllung der Aufgaben gehört die Abgabe und Präsentation eines im Regelfall auf den *Rechnern der RBI* laufenden, kommentierten Programms, sowie eine schriftliche Ausarbeitung, die folgendes enthält:

- eine kurze Zusammenfassung der Aufgabenstellung in eigenen Worten
- eine kurze Erläuterung der Grundlagen, die zur Lösung der Aufgabe notwendig waren.
- eine kurze Beschreibung der Lösung selbst.
- ein Protokoll, das den Prozess der Problemlösung dokumentiert, insbesondere sollte daraus hervor gehen:
  - wie die Arbeit innerhalb der Gruppe verteilt wurde, die einzelnen notwendigen Tätigkeiten sollten dabei möglichst detailliert erfasst werden (z.B. Verstehen der Aufgabenstellung, Entwurf der Implementierung, Realisierung der Implementierung, Anfertigen der Ausarbeitung, Testen der Programme, ...)
  - wie viel Zeit für die einzelnen Tätigkeiten benötigt wurden.
- die Dokumentation (und Interpretation) der durchgeführten Tests.

Die Programme und die Ausarbeitung sollen mit CVS (siehe Abschnitt) verwaltet und spätestens zum jeweiligen Abgabetermin dort eingecheckt sein.

Es wird verschiedene Aufgabenblätter geben, deren Bearbeitungszeit in der Regel 2-4 Wochen beträgt. Die Präsentation der Programme erfolgt zeitnah nach dem Abgabetermin des Aufgabenblattes (Termine werden hierfür individuell mit der jeweiligen Gruppe vereinbart).

1.2 Haskell

Die Implementierung der Programme soll in Haskell erfolgen. Grundkenntnisse in Haskell oder anderen Funktionalen Programmiersprachen sind für die Teilnahme am Praktikum notwendig.

1.2.1 Material zu Haskell

Haskell[1] ist eine der zur Zeit wohl bedeutendsten, nicht-strikten, funktionalen Programmiersprachen. Wer sich eingehender in Haskell vertiefen möchte, sei auf den Haskell-Report (Jones, 2003) verwiesen, in dem die Sprache definiert wird.

Zur Einarbeitung in Haskell sind das deutsche Buch (Chakravarty & Keller, 2004), die Bücher (Thompson, 1999), (Bird, 1998), (O'Sullivan et al., 2008) sowie (Hudak et al., 2000) und auch die Skripte (Schmidt-Schauß, (Wintersemester 2011 / 2012)) und (Schmidt-Schauß, (Sommersemester 2011)) empfehlenswert.

Für Haskell gibt es mittlerweile eine recht große Anzahl von Standardbibliotheken, die hierarchisch organisiert sind. Die Dokumentation der Bibliotheken ist online unter http://www.haskell.org/ghc/docs/latest/html/libraries/index.html zu finden.

Zum Haskell-Programmieren verwenden wir den `GHC`[2].

1.2.2 GHC

Der Glasgow Haskell Compiler[3] hält sich an den Haskell-Report. Im `GHC` gibt es Erweiterungen, die z.B. die Typklassen betreffen. Der `GHC` ist fast vollständig in Haskell geschrieben und erzeugt C-Code als Zwischencode, der dann von einem auf dem System verfügbaren C-Compiler in ein ausführbares Programm übersetzt wird. Diese Tatsache macht den `GHC` äußerst portierbar. Für einige Plattformen gibt es zusätzlich einen Code-Erzeuger. `GHC` bietet zusätzlich noch den `GHCi`, der eine interaktive Version des `GHC` darstellt und als Interpreter zum schnellen Programmieren und Testen verwendet werden kann.

Der `GHCi` wird in einer Konsole mit dem Kommando

```
ghci
```

gestartet (dafür muss der Pfad zur GHC-Installation im Suchpfad der Konsole eingetragen sein). Im Interpreter bietet das Kommando `:help` eine Hilfe und eine Übersicht über die Interpreterkommandos an. Mit `:load` kann eine Quelldatei geladen werden. Alternativ kann dies auch direkt beim Starten des `GHCi` als Parameter durchgeführt werden mit

[1] Die offizielle Homepage zu Haskell ist http://haskell.org.

[2] Ein vollständige Liste aller Haskell-Implementierungen ist unter http://www.haskell.org/haskellwiki/Implementations zu finden.

[3] Die Homepage des `GHC` ist http://haskell.org/ghc.

```
ghci "Dateiname"
```

Haskell Quellcode-Dateien haben die Endung `.hs` oder für Dateien im Literate-Style mit `.lhs`: Im „normalen“ Stil sind alle Zeilen Programme, außer solchen die explizit als Kommentar gekennzeichnet sind (mit “`--`” wird ein Zeilenkommentar markiert, mit “`{-`” und “`-}`” ein geklammerter Kommentar). Hingegen werden im „Literate Haskell“-Stil alle Text-Zeilen als Kommentar interpretiert, es sei denn sie werden explizit als Programm gekennzeichnet. Programmzeilen beginnen mit einen Größerzeichen und einem Leerzeichen (“`> `”), zusätzlich muss sich zwischen Kommentar und Codezeilen immer eine Leerzeile befinden, ansonsten bekommen wir eine Fehlermeldung:

```
.... line 8: unlit: Program line next to comment
phase ‘Literate pre-processor’ failed (exitcode = 1)
```

Zum Compilieren eines Programms mit dem `GHC` muss i.A. ein Modul namens `Main` definiert werden, welches eine Funktion `main` vom Typ `IO ()` enthält (diese Funktion wird beim Programmstart durchgeführt). Ein solches Modul kann dann compiliert werden durch den Aufruf:

```
ghc --make -o "OutputName" Main.hs
```

Dieser Aufruf compiliert die Datei namens `Main.hs` und deren Importe (hierfür dient u.a. der Parameter `--make`) und erstellt eine ausführbare Datei namens `OutputName`.

1.2.3 Haskell Platform

Sofern verfügbar empfiehlt es sich die Haskell Platform http://hackage.haskell.org/platform/ zu installieren. Diese beinhaltet neben dem GHC noch weitere Werkzeuge, wie den Lexer-Generator Alex und den Parser-Generator Happy, sowie einige zusätzliche Programmbibliotheken.

1.2.4 Cabal, Hackage

Neben den Standardbibliotheken gibt es unter http://hackage.haskell.org eine große Sammlung weiterer Pakete, die in einem standardisierten Format vorliegen (so genannte Cabal-Packages).

Unter http://www.haskell.org/haskellwiki/Cabal/How_to_install_a_Cabal_package kann man nachlesen, wie diese installiert werden.

1.3 Quellcode

Der Quellcode sollte *wartbar* sein, und dementsprechend aufgebaut, dokumentiert und kommentiert werden. Es empfiehlt sich die Dateien mithilfe der hierarchischen Modulstruktur zu strukturieren. Zudem sollte durch geeignete Import- und Export-Listen in den Moduldeklarationen eine geeignete Kapselung erfolgen. Im Praktikum wird der Quellcode mittels CVS[4] verwaltet werden (siehe Abschnitt 2.1).

1.4 Dokumentation

Die erstellten Programme sollten ausführlich kommentiert werden, so dass ein „Leser“ des Programms dieses nachvollziehen kann. Zusätzlich bietet es sich an für die Module eine HTML-Dokumentation mit Hilfe des Tools *Haddock*[5] zu erstellen (siehe Abschnitt 2.2). Es sei jedoch angemerkt, dass eine reine Haddock-Dokumentation *nicht* ausreicht, da mit Haddock nur die exportierten Funktionen und Datentypen dokumentiert werden und zudem eher deren *Verwendung* und nicht deren Implementierung erklärt wird.

1.5 Tests

Sämtliche implementierten Funktionen, Datenstrukturen und Module müssen *getestet* werden. Teilweise sind Testdaten bzw. Testaufrufe vorgegeben. Diese müssen durchgeführt und im Idealfall auch bestanden werden. Es ist aber auch notwendig eigene Tests mit sinnvoll überlegten Testaufrufen durchzuführen. Sämtliche Tests sind zu *dokumentieren* und derart zu gestalten, dass sie leicht erneut durchgeführt, d.h. reproduziert, werden können. Zum Testen bietet sich eventuell auch die QuickCheck-Bibliothek (http://hackage.haskell.org/cgi-bin/hackage-scripts/package/QuickCheck) an.

[4] Concurrent Versions System, http://www.nongnu.org/cvs/
[5] http://haskell.org/haddock

2 Hintergrundwissen

In diesem Kapitel sind genauere Erläuterungen zu einigen Themen zu finden. Diese beziehen sich nicht direkt auf einzelne Aufgaben, sondern stellen eher ein notwendiges Grundwissen dar.

2.1 Concurrent Versions System

CVS ist ein Software-System zur Versionsverwaltung von Dateien insbesondere für den Mehrbenutzerbetrieb. Hierbei werden die Dateien zentral auf einem Server in einem so genannten Repository gespeichert. Die Benutzer greifen auf die Dateien über einen CVS-Client zu und erhalten ihre lokale Arbeitskopie. Beim so genannten Einchecken lädt der Benutzer seine geänderten Dateien auf den Server. Wurde zwischenzeitlich die Datei auf dem Server geändert, so versucht CVS die Änderungen nachzuziehen, d.h. die beiden Dateien zu "mergen". Kommt es hierbei zu Konflikten, so muss der Benutzer diese per Hand beheben.

2.1.1 Zugriff per ssh

Um sichere Verbindungen zu verwenden, ist es möglich per `ssh` auf einen CVS-Server zuzugreifen. Bei Unix/Linux-System ist dafür notwendig, dass die Umgebungsvariable `CVS_RSH` auf den Wert `ssh` gesetzt ist. Verwendet man

z.B. die Bash-Shell, so sollte in der Konfigurationsdatei `.bashrc` ein Eintrag der Art

```
export CVS_RSH=ssh
```

stehen. Genauere Informationen zum Zugang zum CVS-Server werden während der ersten Besprechung bekannt gegeben.

2.1.2 Arbeitskopie vom Server holen

Um eine Arbeitskopie vom Repository auf den lokalen Rechner zu laden, sollte man `cvs get` verwenden, wobei der Server, das Repository, den User-Namen und das auszucheckende Modul spezifiziert werden müssen, in der Form:

```
cvs -d user@hostname:repositorypfad get modulname
```

Beim Zugang über `ssh` muss man anschließend sein Passwort eingeben und erhält dann eine Arbeitskopie.

2.1.3 Arbeitskopie lokal aktualisieren

Um eine Arbeitskopie lokal zu aktualisieren, also Änderungen vom Server in die lokale Kopie einzuspielen, kann man `cvs update` benutzen. Es empfiehlt sich die Option `-d` zu verwenden, die auch neu hinzugefügte Verzeichnis herunter lädt. D.h. wenn man sich innerhalb der Arbeitskopie befindet:

```
cvs update -d
```

2.1.4 Dateien einchecken

Um die eigenen Änderungen auf den CVS-Server hochzuladen, ist das Kommando `cvs commit` zu verwenden, welches zusätzlich mit der Option `-m` aufgerufen werden sollte, die es erlaubt noch einen Kommentar bezüglich der Änderungen anzugeben (was wurde geändert, warum). Ruft man `cvs commit` ohne Dateinamen aus, so werden *alle* geänderten Dateien hochgeladen. Noch zwei Beispiele:

```
cvs commit -m "Programm verbessert, Bug XYZ entfernt" datei.hs
```

lädt die Änderungen an der Datei `datei.hs` auf den Server.

```
cd tmp
cvs commit -m ""
```

lädt alle Änderungen an Dateien ab dem Verzeichnis `tmp` auf den CVS-Server.

2.1.5 Hinzufügen von Dateien und Verzeichnissen

Um Dateien oder Verzeichnisse hinzu zu fügen, reicht es *nicht* ein `cvs commit` auf diese Datei zu machen. Man erhält dann die Fehlermeldung

```
cvs commit: nothing known about ‘dateiname’
```

Zum Hinzufügen muss das Kommando `cvs add` verwendet werden. Hiermit werden Verzeichnisse sofort hinzugefügt (allerdings nicht deren Inhalt!). Dateien werden zwar hinzugefügt, aber nach dem Hinzufügen muss noch ein `cvs commit` auf die Datei erfolgen. Ein Beispiel:

```
cvs add tmp/
cvs add datei.hs
cvs commit -m "" datei.hs
```

2.1.6 Keyword-Substitution und Binäre Dateien

Da CVS in Dateien Schlüsselwörter ersetzt (z.B. wird `$Date$` durch das Datum des letzten Eincheckens ersetzt) muss man beim Einchecken von binären Dateien aufpassen, da dort ja kein Ersetzen erfolgen sollte. Hierfür sollte man die Option `-kb` beim `cvs add` verwenden, z.B.

```
cvs add -kb anleitung.pdf
```

Man kann diese Option auch noch später setzen mit `cvs admin` und anschließendem `cvs update`.

2.1.7 Graphische Oberflächen für CVS

Neben den hier vorgestellten Kommandos für die Konsole gibt es zahlreiche graphische Oberflächen zum Benutzen von CVS. Z.B. verfügt die IDE Eclipse (http://www.eclipse.org/) bereits über eingebaute CVS-Unterstützung. Unter KDE gibt es das Programm `cervisia` (http://cervisia.kde.org/), für MS Windows empfiehlt sich das Programm `TortoiseCVS` (http://www.tortoisecvs.org/)

2.2 Haddock – A Haskell Documentation Tool

Haddock (http://haskell.org/haddock) dient zum Erstellen einer HTML-Dokumentation anhand speziell kommentierter Haskell-Quellcode-Dateien. Hierbei wird im Allgemeinen nur für jene Funktionen und Datentypen eine

Dokumentation erstellt, die in der Exportliste eines Moduls vorhanden sind, und – bei Funktionen – explizit mit einer Typsignatur versehen sind.

Wir gehen in diesem Abschnitt auf einige Grundfunktionalitäten von Haddock ein, die vollständige Dokumentation ist auf oben genannter Webseite verfügbar.

2.2.1 Dokumentation einer Funktionsdefinition

Wir betrachten das folgende Beispiel

```
quadrat :: Integer -> Integer
quadrat x = x * x
```

Ein Dokumentationsstring für diese Definition kann wie folgt hinzugefügt werden:

```
-- | Die 'quadrat'-Funktion quadriert Integer-Zahlen
quadrat :: Integer -> Integer
quadrat x = x * x
```

D.h. Haddock-Kommentare beginnen mit `-- |` und müssen vor der Typdeklaration stehen. Man beachte, dass bei Verwendung von Literate Haskell, die Haddock-Kommentare im Code-Teil stehen müssen, d.h. die Definition hat dann die Form

```
> -- | Die 'quadrat'-Funktion quadriert Integer-Zahlen
> quadrat :: Integer -> Integer
> quadrat x = x * x
```

während

```
| Die 'quadrat'-Funktion quadriert Integer-Zahlen


> quadrat :: Integer -> Integer
> quadrat x = x * x
```

nicht funktioniert.

Man beachte, dass Haddock mit sämtlichen Quelldateien aufgerufen werden muss, um eine korrekte Verlinkung der Dokumente zu erhalten. Für Dateien `file1`, ..., `fileN` ist der Haddock-Aufruf

```
haddock ... --html  -o ausgabe file1 ... fileN
```

Hierbei bedeutet `--html`, dass eine HTML-Dokumentation generiert wird, und `-o ausgabe` bedeutet, dass die HTML-Dateien im Verzeichnis `ausgabe` abgelegt werden (dieses muss vorher vorhanden sein!).

2.3 Datentypen und Typklassen in Haskell

In diesem Abschnitt werden in Kurzform selbst definierte Datentypen, Typklassen und die `Record`-Syntax für Datentypen erläutert.

2.3.1 Datentypdefinition

Neben primitiven Datentypen, die Haskell bereits bereitstellt, (z.B. für ganze Zahlen (`Int` für beschränkte Zahlen, `Integer` für unbeschränkte Zahlen) für Fließkommazahlen (`Double` und `Float`), für Bruchzahlen `Rational`) und komplexeren Typen wie Tupel, Listen, Arrays, etc., kann man in Haskell Datentypen selbst definieren. Hierfür gibt es im Wesentlichen drei verschiedene Möglichkeiten:

- Mit `type` kann ein *Typsynonym* definiert werden, d.h. man vergibt einen neuen Namen für schon definierte Typen. Ein einfaches Beispiel ist:

  ```
  type Wahrheitswert = Bool
  ```

  Ein komplexeres Beispiel ist ein Typsynonym für Wörterbücher, welches polymorph über dem Typ der Einträge ist:

  ```
  type Woerterbuch a = [(Integer,a)]
  ```

- Mit `data` wird ein neuer Datentyp definiert. Ein einfacher Aufzählungstyp für die RGB-Farben kann z.B. definiert werden mit

  ```
  data RGB = Rot | Gruen | Blau
  ```

  Hierbei ist `RGB` ein neuer *Typ* und `Rot`, `Gruen`, `Blau` sind neue *Datenkonstruktoren*. Für diese Datenkonstruktoren kann Pattern-Matching verwendet werden. Z.B. kann man eine Funktion definieren, die jede RGB-Farbe in einen String konvertiert:

  ```
  rgbToString Rot = "rot"
  rgbToString Gruen = "gruen"
  rgbToString Blau = "blau"
  ```

  Datentypen können polymorph sein, z.B.

  ```
  entwederOder a b = Links a | Rechts b
  ```

und Datentypen können *rekursiv* sein. Die in Haskell eingebauten Listen sind rekursive Datentypen. Man könnte diese selbst definieren als

```
data Liste a = Nil | Cons a (Liste a)
```

Binäre Bäume mit Blattmarkierungen können rekursiv definiert werden als

```
data Baum a = Blatt a | Knoten (Baum a) (Baum a)
```

- Mit `newtype` wird ein Typsynonym definiert, dass durch einen zusätzlichen Datenkonstruktor verpackt wird. Rein syntaktisch ist `newtype` überflüssig, da dies auch stets durch eine `data`-Deklaration möglich ist. Das Verwenden der `newtype`-Syntax hat den Vorteil, dass der Compiler weiß, dass es sich eigentlich nur um ein verpacktes Typsynonym handelt. Ein weiterer Grund ist, dass für mit `type`-deklarierte Typsynonyme keine Typklasseninstanzen definiert werden können, während dies für mit `newtype` definierte Typen möglich ist.

2.3.2 Typklassen

Typklassen dienen dazu Funktionsnamen und Operatoren zu *überladen*, d.h. den gleichen Funktionsnamen oder Operator für *unterschiedliche* Typen zu verwenden. Z.B. kann deshalb in Haskell den Gleichheitstest == sowohl auf Integerzahlen aber auch auf Listen von Zeichen verwenden. Hierfür ist in Haskell bereits die Typklasse `Eq` wie folgt vordefiniert:

```
class  Eq a where
  (==), (/=) :: a -> a -> Bool

  -- Minimal complete definition:
  --      (==) or (/=)
  x /= y     =  not (x == y)
  x == y     =  not (x /= y)
```

Dieser Code definiert die Klasse `Eq`. Für Typen die Instanz dieser Klasse sind, kann der Gleichheitstest `==` und der Ungleichheitstest `/=` verwendet werden. Innerhalb der Definition sind noch *Default*-Implementierungen für `==` und `/=` angegeben. Beim Instantiieren eines Typs können diese default-Implementierungen überschrieben werden. Für die Klasse `Eq` muss entweder

der Gleichheitstest oder der Ungleichheitstest beim instantiieren überschrieben werden, den jeweils anderen Test erhält quasi „geschenkt“. Die Abhängigkeit von einer Klasseninstanz sieht man auch, wenn man sich den Typ von == im Interpreter anzeigen lässt:

```
>  :t (==)
(==) :: (Eq a) => a -> a -> Bool
```

Die Angaben links vom => sind *Typklassenbeschränkungen*. Das ganze ist zu lesen als, Für alle Typen `a`, die Instanzen der Klasse `Eq` sind, hat == den Typ `a -> a -> Bool`.

Eine Instanz der Klasse `Eq` für den eben definierten Datentypen `RGB` könnte man definieren als:

```
instance Eq (RGB) where
 (==) Rot Rot     = True
 (==) Gruen Gruen = True
 (==) Blau Blau   = True
 (==) _ _         = False
```

Manche Typklasseninstanzen können jedoch auch automatisch aufgrund der Struktur der Datentypdefinition automatisch vom Compiler generiert werden. Hierfür dient das Schlüsselwort `deriving`. Wir hätten schreiben können:

```
data RGB = Rot | Gruen | Blau
 deriving(Eq)
```

und uns damit die Angabe der Typklasseninstanz ersparen können, da sie automatisch generiert wird.

Typklasseninstanzen können nicht für Typsynonyme definiert werden, es sei denn sie wurden mittels `newtype` definiert. Z.B. können wir für den Typ `Wahrheitswert` mit der Definition

```
newtype Wahrheitswert = WW Bool
```

eine Typklasseninstanz für `Eq` definieren:

```
instance Eq Wahrheitswert where
 (==) (WW a) (WW b)  = a == b
```

Hierbei haben wir den Gleichheitstest einfach auf den Gleichheitstest für Boolesche Werte zurück geführt.

Weitere oft verwendete Typklassen sind die Klassen `Show` und `Read`. Daten der Typen Instanz der Klasse `Show` sind können in Strings konvertiert (mit der Funktion `show`) und damit angezeigt werden. Umgekehrt können Strings mit der `read`-Funktion in Daten eines Types konvertiert werden, wenn der Typ Instanz der Klasse `Read` ist.

Da der `GHCi` die `show`-Funktion zum Anzeigen von Werten verwendet, erhält man im Falle, dass keine Klasseninstanz definiert wurde die Fehlermeldung:

```
<interactive>:1:0:
    No instance for (Show (Baum t))
      arising from a use of ‘print’ at <interactive>:1:0-25
    Possible fix: add an instance declaration for (Show (Baum t))
    In a stmt of a ’do’ expression: print it
```

Typklassen bieten auch eine Möglichkeit der Vererbung, d.h. man kann fordern, dass eine Instanz einer Typklasse ABC nur dann erlaubt ist, wenn der Typ bereits Instanz einer anderen Typklasse XYZ ist. Eine solche Klasse ist die Klasse `Num`, die bereits fordert, dass der Typ Instanzen für `Eq` und `Show` ist.

```
class  (Eq a, Show a) => Num a  where
   (+), (-), (*)  :: a -> a -> a
   negate         :: a -> a
   abs, signum    :: a -> a
   fromInteger    :: Integer -> a
```

Dies entspricht quasi einer Mehrfachvererbung: Man kann `Num` als Subklasse der beiden Klassen `Eq` und `Show` auffassen.

Umgekehrt kann man bei der Instanzbildung auch Bedingungen stellen: Wir betrachten als Beispiel eine Instanz der Klasse `Show` für den Typ `Baum` von oben. Wir können polymorphe Bäume nur dann sinnvoll anzeigen, wenn wir bereits wissen wie die Markierungen des Baum angezeigt werden. In der Instanzdefinition können wir dies wie folgt ausdrücken:

```
instance Show a => Show (Baum a) where
 show (Blatt a) = show a
 show (Knoten l r) = "<" ++ show l  ++ "|" ++ show r ++ ">"
```

Diese Bedingung ist zwingend erforderlich, da wir ansonsten den Aufruf `show a` für die Blattmarkierung nicht durchführen dürfen. Lassen wir

`Show a =>` in obiger Definition weg, dann meldet uns der Compiler wie erwartet einen Fehler:

```
Could not deduce (Show a) from the context (Show (Baum a))
  arising from a use of ‘show’ at test.hs:5:18-23
Possible fix:
  add (Show a) to the context of the instance declaration
In the expression: show a
In the definition of ‘show’: show (Blatt a) = show a
In the instance declaration for ‘Show (Baum a)’
```

2.3.3 Record-Syntax für Haskell `data`-Deklarationen

Haskell bietet neben der normalen Definition von Datentypen auch die Möglichkeit eine spezielle Syntax zu verwenden, die insbesondere dann sinnvoll ist, wenn ein Datenkonstruktor viele Argumente hat.

Wir betrachten zunächst den normal definierten Datentypen `Student` als Beispiel:

```
> data Student = Student
>                  Int     -- Matrikelnummer
>                  String  -- Vorname
>                  String  -- Name
>                  String  -- Studiengang
>                  Int     -- Fachsemester
```

Ohne die Kommentare ist nicht ersichtlich, was die einzelnen Komponenten darstellen. Außerdem muss man zum Zugriff auf die Komponenten neue Funktionen definieren. Beispielweise

```
> vorname :: Student -> String
> vorname (mno vorname name stdgang fsem) = vorname
```

Wenn nun Änderungen am Datentyp vorgenommen werden – zum Beispiel eine weitere Komponente für das Hochschulsemester wird hinzugefügt – dann müssen alle Funktionen angepasst werden, die den Datentypen verwenden:

```
> data Student = Student
>                  Int     -- Matrikelnummer
```

```
>                   String  -- Vorname
>                   String  -- Name
>                   String  -- Studiengang
>                   Int     -- Fachsemester
>                   Int     -- Hochschulsemester
>
> vorname :: Student -> String
> vorname (mno vorname name stdgang fsem hsem) = vorname
```

Um diese Nachteile zu vermeiden, bietet es sich an, die Record-Syntax zu verwenden. Diese erlaubt zum einen die einzelnen Komponenten mit Namen zu versehen:

```
> data Student = Student {
>                   matrikelnummer :: Int,
>                   vorname        :: String,
>                   name           :: String,
>                   studiengang    :: String,
>                   fachsemester   :: Int
>                  }
```

Eine konkrete Instanz würde mit der normalen Syntax initialisiert mittels

```
Student 1234567 "Hans" "Mueller" "Informatik" 5
```

Für den Record-Typen ist dies genauso möglich, aber es gibt auch die Möglichkeit die Namen zu verwenden:

```
Student{matrikelnummer=1234567,
        vorname="Hans",
        name="Mueller",
        studiengang="Informatik",
        fachsemester=5}
```

Hierbei spielt die Reihenfolge der Einträge keine Rolle, z.B. ist

```
Student{fachsemester=5,
        vorname="Hans",
        matrikelnummer=1234567,
        name="Mueller",
        studiengang="Informatik"
       }
```

genau dieselbe Instanz.

Zugriffsfunktionen für die Komponenten brauchen nicht zu definiert werden, diese sind sofort vorhanden und tragen den Namen der entsprechenden Komponente. Z.B. liefert die Funktion `matrikelnummer` angewendet auf eine Student-Instanz dessen Matrikelnummer. Wird der Datentyp jetzt wie oben erweitert, so braucht man im Normalfall wesentlich weniger Änderungen am bestehenden Code.

Die Schreibweise mit Feldnamen darf auch für das Pattern-Matching verwendet werden. Hierbei müssen nicht alle Felder spezifiziert werden. So ist z.B. eine Funktion die testet, ob der Student einen Nachnamen beginnend mit `'A'` hat implementierbar als

```
> nachnameMitA Student{nachname = 'A':xs} = True
> nachnameMitA _ = False
```

Diese Definition ist äquivalent zur Definition

```
> nachnameMitA Student _ _ ('A':xs) _ _ = True
> nachnameMitA _                       = False
```

2.4 Modularisierung in Haskell

Module dienen zur

Strukturierung / Hierarchisierung: Einzelne Programmteile können innerhalb verschiedener Module definiert werden; eine (z. B. inhaltliche) Unterteilung des gesamten Programms ist somit möglich. Hierarchisierung ist möglich, indem kleinere Programmteile mittels Modulimport zu größeren Programmen zusammen gesetzt werden.

Kapselung: Nur über Schnittstellen kann auf bestimmte Funktionalitäten zugegriffen werden, die Implementierung bleibt verdeckt. Sie kann somit unabhängig von anderen Programmteilen geändert werden, solange die Funktionalität (bzgl. einer vorher festgelegten Spezifikation) erhalten bleibt.

Wiederverwendbarkeit: Ein Modul kann für verschiedene Programme benutzt (d.h. importiert) werden.

2.4.1 Module in Haskell

In einem Modul werden Funktionen, Datentypen, Typsynonyme, usw. definiert. Durch die Moduldefinition können diese exportiert Konstrukte werden, die dann von anderen Modulen importiert werden können.

Ein Modul wird mittels

$$\texttt{module}\ \mathit{Modulname}(\mathit{Exportliste})\ \texttt{where}$$
$$\left.\begin{array}{l}\mathit{Modulimporte},\\ \mathit{Datentypdefinitionen},\\ \mathit{Funktionsdefinitionen},\ldots\end{array}\right\}\ \mathit{Modulrumpf}$$

definiert. Hierbei ist `module` das Schlüsselwort zur Moduldefinition, *Modulname* der Name des Moduls, der mit einem Großbuchstaben anfangen muss. In der *Exportliste* werden diejenigen Funktionen, Datentypen usw. definiert, die durch das Modul exportiert werden, d.h. von außen sichtbar sind.

Für jedes Modul muss eine separate Datei angelegt werden, wobei der Modulname dem Dateinamen ohne Dateiendung entsprechen muss.

Ein Haskell-Programm besteht aus einer Menge von Modulen, wobei eines der Module ausgezeichnet ist, es muss laut Konvention den Namen `Main` haben und eine Funktion namens `main` definieren und exportieren. Der Typ von `main` ist auch per Konvention festgelegt, er muss `IO ()` sein, d.h. eine Ein-/Ausgabe-Aktion, die nichts (dieses „Nichts“ wird durch das Nulltupel `()` dargestellt) zurück liefert. Der Wert des Programms ist dann der Wert, der durch `main` definiert wird. Das Grundgerüst eines Haskell-Programms ist somit von der Form:

```
module Main(main) where
 ...
 main = ...
 ...
```

Im folgenden werden wir den Modulexport und -import anhand folgendes Beispiels verdeutlichen:

Beispiel 2.4.1.

```
module Spiel where
  data Ergebnis = Sieg | Niederlage | Unentschieden
  berechneErgebnis a b = if a > b then Sieg
```

```
                        else if a < b then Niederlage
                             else Unentschieden
istSieg Sieg = True
istSieg _    = False
istNiederlage Niederlage = True
istNiederlage _          = False
```

2.4.1.1 Modulexport

Durch die *Exportliste* bei der Moduldefinition kann festgelegt werden, was exportiert wird. Wird die Exportliste einschließlich der Klammern weggelassen, so werden alle definierten, bis auf von anderen Modulen importierte, Namen exportiert. Für Beispiel 2.4.1 bedeutet dies, dass sowohl die Funktionen `berechneErgebnis`, `istSieg`, `istNiederlage` als auch der Datentyp `Ergebnis` samt aller seiner Konstruktoren `Sieg`, `Niederlage` und `Unentschieden` exportiert werden. Die Exportliste kann folgende Einträge enthalten:

- Ein Funktionsname, der im Modulrumpf definiert oder von einem anderem Modul importiert wird. Operatoren, wie z.B. + müssen in der Präfixnotation, d.h. geklammert `(+)` in die Exportliste eingetragen werden.

  Würde in Beispiel 2.4.1 der Modulkopf

  ```
  module Spiel(berechneErgebnis) where
  ```

  lauten, so würde nur die Funktion `berechneErgebnis` durch das Modul `Spiel` exportiert.

- Datentypen die mittels `data` oder `newtype` definiert wurden. Hierbei gibt es drei unterschiedliche Möglichkeiten, die wir anhand des Beispiels 2.4.1 zeigen:
  - Wird nur `Ergebnis` in die Exportliste eingetragen, d.h. der Modulkopf würde lauten

    ```
    module Spiel(Ergebnis) where
    ```

    so wird der Typ `Ergebnis` exportiert, nicht jedoch die Datenkonstruktoren, d.h. `Sieg`, `Niederlage`, `Unentschieden` sind von außen nicht sichtbar bzw. verwendbar.
  - Lautet der Modulkopf

    ```
    module Spiel(Ergebnis(Sieg, Niederlage))
    ```

so werden der Typ `Ergebnis` und die Konstruktoren `Sieg` und `Niederlage` exportiert, nicht jedoch der Konstruktor `Unentschieden`.

- Durch den Eintrag `Ergebnis(..)`, wird der Typ mit sämtlichen Konstruktoren exportiert.

- Typsynonyme, die mit `type` definiert wurden, können exportiert werden, indem sie in die Exportliste eingetragen werden, z.B. würde bei folgender Moduldeklaration

```
module Spiel(Result) where
  ... wie vorher ...
  type Result = Ergebnis
```

der mittels `type` erzeugte Typ `Result` exportiert.

- Schließlich können auch alle exportierten Namen eines importierten Moduls wiederum durch das Modul exportiert werden, indem man `module` *Modulname* in die Exportliste aufnimmt, z.B. seien das Modul `Spiel` wie in Beispiel 2.4.1 definiert und das Modul `Game` als:

```
module Game(module Spiel, Result) where
  import Spiel
  type Result = Ergebnis
```

Das Modul `Game` exportiert alle Funktionen, Datentypen und Konstruktoren, die auch `Spiel` exportiert sowie zusätzlich noch den Typ `Result`.

2.4.1.2 Modulimport

Die exportierten Definitionen eines Moduls können mittels der `import` Anweisung in ein anderes Modul importiert werden. Diese steht am Anfang des Modulrumpfs. In einfacher Form geschieht dies durch

`import` *Modulname*

Durch diese Anweisung werden sämtliche Einträge der Exportliste vom Modul mit dem Namen *Modulname* importiert, d.h. sichtbar und verwendbar.

Will man nicht alle exportierten Namen in ein anderes Modul importieren, so ist dies auf folgende Weisen möglich:

Explizites Auflisten der zu importierenden Einträge: Die importierten Namen werden in Klammern geschrieben aufgelistet. Die Einträge werden hier genauso geschrieben wie in der Exportliste.

Z.B. importiert das Modul

```
module Game where
  import Spiel(berechneErgebnis, Ergebnis(..))
  ...
```

nur die Funktion `berechneErgebnis` und den Datentyp `Ergebnis` mit seinen Konstruktoren, nicht jedoch die Funktionen `istSieg` und `istNiederlage`.

Explizites Ausschließen einzelner Einträge: Einträge können vom Import ausgeschlossen werden, indem man das Schlüsselwort `hiding` gefolgt von einer Liste der ausgeschlossen Einträge benutzt.

Den gleichen Effekt wie beim expliziten Auflisten können wir auch im Beispiel durch Ausschließen der Funktionen `istSieg` und `istNiederlage` erzielen:

```
module Game where
  import Spiel hiding(istSieg,istNiederlage)
  ...
```

Die importierten Funktionen sind sowohl mit ihrem (unqualifizierten) Namen ansprechbar, als auch mit ihrem qualifizierten Namen: *Modulname*.*unqualifizierter Name*, manchmal ist es notwendig den qualifizierten Namen zu verwenden, z.B.

```
module A(f) where
  f a b = a + b

module B(f) where
  f a b = a * b

module C where
  import A
  import B
  g = f 1 2 + f 3 4 -- funktioniert nicht
```

führt zu einem Namenskonflikt, da `f` mehrfach (in Modul `A` und `B`) definiert wird.

```
Prelude> :l C.hs

ERROR C.hs:4 - Ambiguous variable occurrence "f"
*** Could refer to: B.f A.f
```

Werden qualifizierte Namen benutzt, wird die Definition von g eindeutig:

```
module C where
  import A
  import B
  g = A.f 1 2 + B.f 3 4
```

Durch das Schlüsselwort `qualified` sind nur die qualifizierten Namen sichtbar:

```
module C where
  import qualified A
  g = f 1 2   -- f ist nicht sichtbar

Prelude> :l C.hs

ERROR C.hs:3 - Undefined variable "f"
```

Man kann auch *lokale Aliase* für die zu importierenden Modulnamen angeben, hierfür gibt es das Schlüsselwort `as`, z.B.

```
import LangerModulName as C
```

Eine durch `LangerModulName` exportierte Funktion `f` kann dann mit `C.f` aufgerufen werden.

Abschließend eine Übersicht: Angenommen das Modul `M` exportiert `f` und g, dann zeigt die folgende Tabelle, welche Namen durch die angegebene `import`-Anweisung sichtbar sind:

Import-Deklaration	definierte Namen
`import M`	`f, g, M.f, M.g`
`import M()`	keine
`import M(f)`	`f, M.f`
`import qualified M`	`M.f, M.g`
`import qualified M()`	keine
`import qualified M(f)`	`M.f`
`import M hiding ()`	`f, g, M.f, M.g`
`import M hiding (f)`	`g, M.g`
`import qualified M hiding ()`	`M.f, M.g`
`import qualified M hiding (f)`	`M.g`
`import M as N`	`f, g, N.f, N.g`
`import M as N(f)`	`f, N.f`
`import qualified M as N`	`N.f, N.g`

2.4.1.3 Hierarchische Modulstruktur

Diese Erweiterung ist nicht durch den Haskell-Report festgelegt, wird jedoch von `GHC` und `Hugs` unterstützt[1]. Sie erlaubt es Modulnamen mit Punkten zu versehen. So kann z.B. ein Modul `A.B.C` definiert werden. Allerdings ist dies eine rein syntaktische Erweiterung des Namens und es besteht nicht notwendigerweise eine Verbindung zwischen einem Modul mit dem Namen `A.B` und `A.B.C`.

Die Verwendung dieser Syntax hat lediglich Auswirkungen wie der Interpreter nach der zu importierenden Datei im Dateisystem sucht: Wird `import A.B.C` ausgeführt, so wird das Modul `A/B/C.hs` geladen, wobei `A` und `B` Verzeichnisse sind.

Die „Haskell Hierarchical Libraries[2]“ sind mithilfe der hierarchischen Modulstruktur aufgebaut, z.B. sind Funktionen, die auf Listen operieren, im Modul `Data.List` definiert.

Damit der `ghci` die Module auch findet, muss das oberste Verzeichnis der Modulstruktur für ihn auffindbar sein. Dafür gibt es beim `ghci` den Parameter

```
-i<dir>         Search for imported modules in the directory <dir>.
```

Wenn wir z.B. gerade im Verzeichnis `Verzeichnis1/Verzeichnis2/` sind

[1] An der Standardisierung der hierarchischen Modulstruktur wird gearbeitet, siehe `http://www.haskell.org/hierarchical-modules`

[2] siehe `http://www.haskell.org/ghc/docs/latest/html/libraries`

und wollen das Modul `Verzeichnis1.Verzeichnis2.Datei` mit Dateinamen `Datei.lhs` laden, so sollten wir `ghci` wie folgt aufrufen:

```
ghci -i:../../ Datei.lhs
```

2.5 Debugging

Haskell stellt nicht viele Tools zum Debuggen zur Verfügung. Der Hauptgrund dafür ist das mächtige statische Typsystem, welches bereits zur Compilezeit viele Programmierfehler erkennen lässt. Trotzdem geben wir hier einige Hinweise zum Debuggen.

In der Standardbibliothek `Debug.Trace` findet sich die Funktion `trace` vom Typ

```
> trace :: String -> a -> a
```

Wenn diese aufgerufen wird, gibt sie den String des ersten Arguments aus und liefert dann als Ergebnis das zweite Argument. Mit dieser Funktion kann man Zwischenwerte beim Berechnen zum Debuggen ausgeben. In Verbindung mit Guards kann man dies relativ komfortabel bewerkstelligen, ohne den Code mit vielen `trace`-Aufrufe zu verseuchen. Wir betrachten ein Beispiel. Sei `f` definiert als

```
> f x y z = e
```

wobei `e` irgendeinen Code darstellt. Um nun zum Debugging, bei jedem Aufruf von `f` die Werte der Argumente auszugeben, kann man wie folgt vorgehen:

```
> f x y z
>   | trace (show x ++ show y ++ show y) False = undefined
>   | otherwise      = e
```

Der erste Guard wird immer ausgewertet und deshalb werden mittels `trace` die Argumente ausgedruckt. Da `trace` aber insgesamt stets `False` liefert trifft der Guard nicht zu und der `otherwise`-Guard wird immer aufgerufen. Will man den `trace`-Aufruf entfernen, so reicht es die zweite Zeile auszukommentieren:

```
> f x y z
>   --  | trace (show x ++ show y ++ show y) False = undefined
>   | otherwise      = e
```

2.6 I/O in Haskell

In einer rein funktionalen Programmiersprache mit verzögerter Auswertung wie Haskell sind Seiteneffekte zunächst verboten. Fügt man Seiteneffekte einfach hinzu (z.B. durch eine „Funktion“ `getZahl`), die beim Aufruf eine Zahl vom Benutzer abfragt und anschließend mit dieser Zahl weiter auswertet, so erhält man einige unerwünschte Effekte der Sprache, die man im Allgemeinen nicht haben möchte.

- Rein funktionale Programmiersprachen sind *referentiell transparent*, d.h. eine Funktion angewendet auf gleiche Werte, ergibt stets denselben Wert im Ergebnis. Die referentielle Transparenz wird durch eine Funktion wie `getZahl` verletzt, da `getZahl` je nach Ablauf unterschiedliche Werte liefert.

- Ein weiteres Gegenargument gegen das Einführen von primitiven Ein-/Ausgabefunktionen besteht darin, dass übliche (schöne) mathematische Gleichheiten wie $e + e = 2 * e$ für alle Ausdrücke der Programmiersprache nicht mehr gelten. Setze `getZahl` für e ein, dann fragt $e * e$ zwei verschiedene Werte vom Benutzer ab, während $2 * e$ den Benutzer nur einmal fragt. Würde man also solche Operationen zulassen, so könnte man beim Transformieren innerhalb eines Compilers übliche mathematische Gesetze nur mit Vorsicht anwenden.

- Durch die Einführung von direkten I/O-Aufrufen besteht die Gefahr, dass der Programmierer ein anderes Verhalten vermutet, als sein Programm wirklich hat. Der Programmierer muss die verzögerte Auswertung von Haskell beachten. Betrachte den Ausdruck `length [getZahl, getZahl]`, wobei `length` die Länge einer Liste berechnet als

  ```
  length [] = 0
  length (_:xs) = 1 + length xs
  ```

  Da die Auswertung von `length` die Listenelemente gar nicht anfasst, würde obiger Aufruf, keine `getZahl`-Aufrufe ausführen.

- In reinen fmonaunktionalen Programmiersprachen wird oft auf die Festlegung einer genauen Auswertungsreihenfolge verzichtet, um Optimierungen und auch Parallelisierung von Programmen durchzuführen.

Z.B. könnte ein Compiler bei der Auswertung von $e_1 + e_2$ zunächst e_2 und danach e_1 auswerten. Werden in den beiden Ausdrücken direkte I/O-Aufrufe benutzt, spielt die Reihenfolge der Auswertung jedoch eine Rolle, da sie die Reihenfolge der I/O-Aufrufe wider spiegelt.

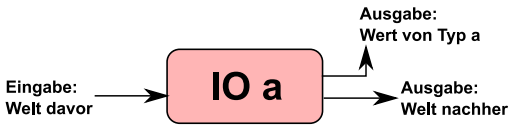
Aus all den genannten Gründen, wurde in Haskell ein anderer Weg gewählt. I/O-Operationen werden mithilfe des so genannten *monadischen I/O* programmiert. Hierbei werden I/O-Aufrufe vom funktionalen Teil gekapselt. Zu Programmierung steht der Datentyp `IO a` zu Verfügung. Ein Wert vom Typ `IO a` stellt jedoch kein Ausführen von Ein- und Ausgabe dar, sondern eine *I/O-Aktion*, die erst beim *Ausführen* (außerhalb der funktionalen Sprache) Ein-/Ausgaben durchführt und anschließend einen Wert vom Typ `a` liefert.

D.h. man setzt innerhalb des Haskell-Programms I/O-Aktionen zusammen. Die große (durch `main`) definierte I/O-Aktion wird im Grunde dann außerhalb von Haskell ausgeführt.

Eine anschauliche Vorstellung dabei ist die folgende. Eine I/O-Aktion ist eine Funktion, die als Eingabe einen Zustand der Welt (des Rechners) erhält und als Ausgabe den veränderten Zustand der Welt sowie ein Ergebnis liefert. Als Haskell-Typ geschrieben:

```
type IO a = Welt -> (a,Welt)
```

Man kann dies auch durch folgende Grafik illustrieren:



Aus Sicht von Haskell sind Objekte vom Typ `IO a` bereits *Werte*, d.h. sie können nicht weiter ausgewertet werden. Dies passt dazu, dass auch andere Funktionen Werte in Haskell sind. Allerdings im Gegensatz zu „normalen“ Funktionen kann Haskell kein Argument vom Typ „Welt“ bereit stellen. Die Ausführung der Funktion geschieht erst durch das Laufzeitsystem, welche die Welt auf die durch `main` definierte I/O-Aktion anwendet.

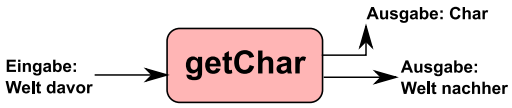
Um nun I/O-Aktionen in Haskell zu Programmieren werden zwei Zutaten benötigt: Zum Einen benötigt man (primitive) Basisoperationen, zum Anderen benötigt man Operatoren, um aus kleinen I/O-Aktionen größere zu konstruieren.

2.6.1 Primitive I/O-Operationen

Wir gehen zunächst von zwei Basisoperationen aus, die Haskell primitiv zur Verfügung stellt. Zum Lesen eines Zeichens vom Benutzer gibt es die Funktion `getChar`:

```
getChar :: IO Char
```

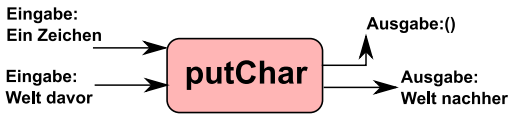
In der Welt-Sichtweise ist `getChar` eine Funktion, die eine Welt erhält und als Ergebnis eine veränderte Welt sowieso ein Zeichen liefert. Man kann dies durch folgendes Bild illustrieren:



Analog dazu gibt es die primitive Funktion `putChar`, die als Eingabe ein Zeichen (und eine Welt) erhält und nur die Welt im Ergebnis verändert. Da alle I/O-Aktionen jedoch noch ein zusätzliches Ergebnis liefern müssen, wird hier der 0-Tupel `()` verwendet.

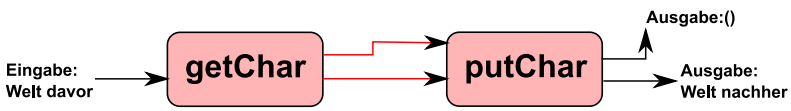
```
putChar :: Char -> IO ()
```

Auch `putChar` lässt sich mit einem Bild illustrieren:



2.6.2 Komposition von I/O-Aktionen

Um aus den primitiven I/O-Aktionen größere Aktionen zu erstellen, werden Kombinatoren benötigt, um I/O-Aktionen miteinander zu verknüpfen. Z.B. könnte man zunächst mit `getChar` ein Zeichen lesen, welches anschließend mit `putChar` ausgegeben werden soll. Im Bild dargestellt möchte man die beiden Aktionen `getChar` und `putChar` wiefolgt sequentiell ausführen und dabei die Ausgabe von `getChar` als Eingabe für `putChar` benutzen (dies gilt sowohl für das Zeichen, aber auch für den Weltzustand):



Genau diese Verknüpfung leistet der Kombinator >>=, der „bind“ ausgesprochen wird. Der Typ des Kombinators ist:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

D.h. er erhält eine IO-Aktion, die einen Wert vom Typ `a` liefert, und eine Funktion die einen Wert vom Typ `a` verarbeiten kann, indem sie als Ergebnis eine IO-Aktion vom Typ `IO b` erstellt.

Wir können nun die gewünschte IO-Aktion zum Lesen und Asgegeben eines Zeichens mithilfe von >>= definieren:

```
echo :: IO ()
echo = getChar >>= putChar
```

Ein andere Variante stellt der >>-Operator (gesprochen: „then“) dar. Er wird benutzt, um aus zwei I/O-Aktionen die Sequenz beider Aktionen zur erstellen, wobei das Ergebnis der ersten Aktion *nicht* von der zweiten Aktion benutzt wird (die Welt wird allerdings weitergereicht). Der Typ von >> ist:

```
(>>) :: IO a -> IO b -> IO b
```

Allerdings muss der Kombinator >> nicht primitiv zur Verfügung gestellt werden, da er leicht mithilfe von >>= definiert werden kann:

```
(>>) :: IO a -> IO b -> IO b
(>>) akt1 akt2 = akt1 >>= \_ -> akt2
```

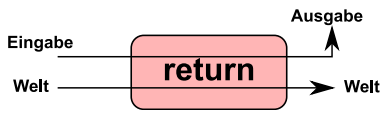
Mithilfe der beiden Operatoren kann man z.B. eine IO-Aktion definieren, die ein gelesenes Zeichen zweimal ausgibt:

```
echoDup :: IO ()
echoDup = getChar >>= (\x -> putChar x >> putChar x)
```

Angenommen wir möchten eine IO-Aktion erstellen, die zwei Zeichen liest und diese anschließend als Paar zurück gibt. Dann benötigen wir eine weitere Operation, um einen beliebigen Wert (in diesem Fall das Paar von Zeichen) in eine IO-Aktion zu verpacken, die nichts anderes tut, als das Paar zu liefern (die Welt wird einfach von der Eingabe zur Ausgabe weitergereicht). Dies leistet die primitive Funktion `return` mit dem Typ

```
return :: a -> IO a
```

Als Bild kann man sich die `return`-Funktion wie folgt veranschaulichen:



Die gewünschte Operation, die zwei Zeichen liest und diese als Paar zurück liefert kann nun definiert werden:

```
getTwoChars :: IO (Char,Char)
getTwoChars = getChar >>= \x ->
              getChar >>= \y ->
              return (x,y)
```

Das Verwenden von `>>=` und dem nachgestellten „`\x -> ...`“-Ausdruck kann man auch lesen als: Führe `getChar` durch, und binde das Ergebnis an `x` usw. Deswegen gibt es als syntaktischen Zucker die `do`-Notation. Unter Verwendung der `do`-Notation erhält die `getTwoChars`-Funktion folgende Definition:

```
getTwoChars :: IO (Char,Char)
getTwoChars = do {
                x <- getChar;
                y <- getChar;
                return (x,y)}
```

Diese Schreibweise ähnelt nun sehr der imperativen Programmierung. Die `do`-Notation ist jedoch nur syntaktischer Zucker, sie kann mit `>>=` wegkodiert werden unter Verwendung der folgenden Regeln:

```
do { x<-e; s }  = e >>= \x -> do { s }
do { e; s }     = e >> do { s }
do { e }        = e
```

Als Anmerkung sei noch erwähnt, dass wir im Folgenden die geschweifte Klammerung und die Semikolons nicht verwenden, da diese durch Einrückung der entsprechenden Kodezeilen vom Parser automatisch eingefügt werden.

Als Fazit zur Implementierung von IO in Haskell kann man sich merken, dass neben den primitiven Operationen wie `getChar` und `putChar`, die Kombinatoren `>>=` und `return` ausreichen, um genügend viele andere Operationen zu definieren.

Wir zeigen noch, wie man eine ganze Zeile Text einlesen kann, indem man `getChar` wiederholt rekursiv aufruft:

```
getLine :: IO [Char]
getLine = do c <- getChar;
             if c == '\n' then
               return []
             else
               do
                cs <- getLine
                return (c:cs)
```

2.6.3 Einige gebräuchliche IO-Funktionen

Haskell bietet in der Prelude (und hauptsächlich in der Bibliothek `System.IO`) jede Menge monadische Funktionen. Für die Ein- und Ausgabe auf dem Bildschirm empfehlen sich die Funktionen

- `getChar :: IO Char`: Lesen eines Zeichens von der Standardeingabe
- `getLine :: IO Char`: Lesen einer Zeile von der Standardeingabe
- `putChar :: Char -> IO ()`: Drucken eines Zeichens auf die Standardausgabe
- `putStr :: String -> IO ()`: Drucken eines Strings auf die Standardausgabe
- `putStrLn :: String -> IO ()`: Drucken eines Strings und anschließendem Zeilenende auf die Standardausgabe
- `print :: (Show a) => a -> IO ()` Anzeigen eines Types und anschließendes Ausdrucken, (`print` ist definiert als `print a = putStrLn (show a)`)

Für die Dateibehandlung bietet sich im Wesentlichen an:

- `readFile :: FilePath -> IO String`: (Verzögertes) Lesen einer Datei (`FilePath` ist ein Synonym für `String` und bezeichnet den Dateinamen (einschließlich eines Pfades))
- `writeFile :: FilePath -> String -> IO ()`: Schreiben eines Strings in eine Datei

- `appendFile :: FilePath -> String -> IO ()` Anhängen eines Strings an eine bereits bestehende Datei

Beachte, dass es mittels `readFile` und `writeFile` im Allgemeinen nicht ohne Weiteres möglich ist, zunächst eine Datei zu lesen und anschließend in die gleiche Datei zu schreiben, da das Lesen den Zugriff auf die Datei solange blockiert, bis die gesamte Datei gelesen wurde. Man kann dies jedoch durch explizites Öffnen und Schließen der Dateihandles umgehen (siehe Dokumentation der Bibliothek `System.IO`).

2.6.4 Monaden

Bisher haben wir zwar erwähnt, dass Haskell *monadisches* IO verwendet. Wir sind jedoch noch nicht darauf eingegangen, warum dieser Begriff verwendet wird. Der Begriff *Monade* stammt aus dem Gebiet der Kategorientheorie (aus der Mathematik). Eine Monade besteht aus einem Typkonstruktor `M` und zwei Operationen:

```
(>>=)  :: M a -> (a -> M b) -> M b
return :: a -> M a
```

wobei zusätzlich die folgenden drei Gesetze gelten müssen.

(1) `return x >>= f  = f x`

(2) `m >>= return  = m`

(3)
```
m1 >>= (\x -> m2 >>= (\y -> m3))
                =
(m1 >>= (\x -> m2)) >>= (\y -> m3)
```

Da Monaden die Sequentialisierung erzwingen, ergibt sich, dass die Implementierung des IO in Haskell sequentialisierend ist, was i.A. gewünscht ist.

Eine der wichtigsten Eigenschaften des monadischen IOs in Haskell ist, dass es keinen Weg aus einer Monade heraus gibt. D.h. es gibt keine Funktion `f:: IO a -> a`, die aus einem in einer I/O-Aktion verpackten Wert nur diesen Wert extrahiert. Dies erzwingt, dass man `IO` nur innerhalb der Monade programmieren kann und i.A. rein funktionale Teile von der I/O-Programmierung trennen sollte.

Dies ist zumindest in der Theorie so. In der Praxis stimmt obige Behauptung nicht mehr, da alle Haskell-Compiler eine Möglichkeit bieten, die IO-Monade zu „knacken“.

Literatur

Bird, R. S. (**1998**). *Introduction to Functional Programming Using Haskell*. Prentice-Hall.

Chakravarty, M. M. T. & Keller, G. C. (**2004**). *Einführung in die Programmierung mit Haskell*. Pearson Studium.

Hudak, P., Peterson, J., & Fasel, J. H. (**2000**). A Gentle Introduction to Haskell. Online verfügbar unter http://haskell.org/tutorial/.

Jones, S. P., editor (**2003**). *Haskell 98 Language and Libraries*. Cambridge University Press. Auch online verfügbar unter http://haskell.org/definition.

O'Sullivan, B., Stewart, D., & Goerzen, J. (**2008**). *Real World Haskell*. O'Reilly Media, Inc. Webseite zum Buch: http://book.realworldhaskell.org/read/.

Schmidt-Schauß, M. (**(Sommersemester 2011)**). Skript zur Vorlesung „Grundlagen der Programmierung 2". http://www.informatik.uni-frankfurt.de/∼prg2/SS2011/index.html. Kapitel 1 + 2.

Schmidt-Schauß, M. (**(Wintersemester 2011 / 2012)**). Skript zur Vorlesung „Einführung in die Funktionale Programmierung". http://www.ki.informatik.uni-frankfurt.de/lehre/WS2011/EFP/.

Thompson, S. (**1999**). *Haskell – The Craft of Functional Programming*. Addison-Wesley.