Prof. Dr. Manfred Schmidt-Schauß
Künstliche Intelligenz und Softwaretechnologie

Fachbereich Informatik (20)
Johann-Wolfgang-Goethe-Universität Frankfurt am Main

Übung: Funktionale Programmierung II

Wintersemester 1996/97

Sven Eric Panitz, Marko Schütz, Frank Kiesel, Norbert Klose, Matthias Mann, M. Schmidt-Schauß

25. Oktober 1996

Inhaltsverzeichnis

1 Organisation

1.1 Sprechstunden

Während des Semesters wird Matthias Mann eine Sprechstunde haben, zu denen er bei Fragen oder Schwierigkeiten im Raum 026 zur Verfügung steht.

Matthias Mann	Dienstag	14-16 Uhr

In sonstigen dringenden Fällen sind wir über e-mail erreichbar:

Matthias Mann	mann@informatik.uni-frankfurt.de
Sven Eric Panitz	panitz@informatik.uni-frankfurt.de

In weiteren Notfällen ist Sven Eric Panitz in Raum 309 erreichbar.

1.2 Computerraum

Ebenfalls zu den Sprechzeiten, wird der Raum 026 im Untergeschoß für Studenten der Vorlesung reserviert sein. Zusätzlich stehen natürlich auch die Rechner in den Fischerräumen zur Verfügung, so daß, wie ich denke, es zu keinen Rechnerengpässen kommen sollte.

1.3 Scheinvergabe

Die Scheine für die Vorlesung werden für ein regelmäßiges und bemühtes Mitarbeiten vergeben, was in diesem Falle, ein wöchentliches Bemühen die Übungsblätter zu lösen, bedeutet.

2 Programmiersprachen

Die Vorlesung ist als Fortsetzung der Vorlesung funktionale Programmierung im WS 95/96 gedacht. Sie kann aber unabhängig von der Vorlesung 'funktionale Programmierung' verstanden werden. Evtl. notwendige Grundlagen werden wiederholt. Dazu gehört die Programmierung in Haskell, einer lazy funktionalen Sprache mit statischer Typung, da in der Vorlesung hauptsächlich die Analyse dieser Sprachen behandelt wird. Daher stehen uns für die Übung bisher das Haskell User Gofer System (hugs), sowie der Haskell B Compiler (HBC) zur Verfügung.

2.1 Hugs

Hugs und dessen Vorgänger Gofer gibt es für verschiedene Rechner und Betriebssysteme: darunter DOS und Windows oder Unix Rechner, wie die HPs hier an der Uni und die Linux PCs, die einige möglicherweise verwenden.
Auf den HPs ist Hugs so installiert, daß mit dem Kommando `hugs` am Shell-Prompt der Interpreter gestartet wird.
Hugs (`http://www.cs.nott.ac.uk/Department/Staff/mpj/hugs.html`) implementiert im Gegensatz zu Gofer — bis auf Schwierigkeiten mit dem Modulsystem — vollständig den Haskell 1.3 Standard, einzusehen in `http://haskell.cs.yale.edu/haskell-report/haskell-report.html`.

2.2 HBC

Der Haskell B Compiler hält sich an den Haskell-Report, erweitert diesen aber auf verschiedene (oft syntaktische) Arten. Z. B. sind im HBC Identifier zulässig, die länderspezifische Zeichen verwenden. Der HBC ist aus dem LML Compiler (s.u.) entwickelt worden und ist auf vielen Platformen, ebenso wie LML, verfügbar.
Der HBC ist auf dem RBI installiert und wird mit `hbc` aufgerufen. Hat man ein Programm, das aus mehreren Modulen besteht, so kann man sich den Skript

hbcmake bedienen, das dann die Reihenfolge der notwendig zu kompelierenden Module steuert. Es gibt auch eine Linux-Version und eine Windows95-Version vom HBC.

2.3 Weitere Sprachen

Wir wollen hier eine kurze Übersicht über die verschiedenen funktionalen Sprachen und ihre Implementierungen geben. Weitere Informationen lassen sich auf dem Netz unter
`http://www.cs.nott.ac.uk/Department/Staff/mpj/faq.html#Haskell`
finden.

Lisp Lisp findet seit Jahrzehnten besonders in der KI-Gemeinde Verwendung. Es ist eine strikte dynamisch getypte Sprache höherer Ordung. Es gibt eine Vielzahl freier und kommerzieller Implementierungen von Lisp. Das u.a. in der Vorlesung des letzten Jahres benutzte Scheme ist auch ein Lisp-Dialekt.

ML ML steht für Meta Language [MTH90] und ist eine strikte funktionale Sprache höherer Ordnung mit dem statischen Typsystem von Milner. ML ist am RBI nicht installiert, was aber bei entsprechenden Bedarf geändert werden kann.

Miranda [1] Miranda ist eine kommerziell verkaufte Programmiersprache, die in vieler Hinsicht Gofer bzw. Haskell ähnelt. Miranda ist eine der ältesten nicht-strikten funktionalen Sprachen und wird deshalb in sehr vielen Lehrbüchern verwendet. Miranda steht auf den RBI-Rechnern leider nicht zur Verfügung. Es gibt auch eine Linux-Version von Miranda. Sie kostet allerdings eine Stange Geld.

LML LML steht für Lazy ML und implementiert die Programmiersprache ML nicht-strikt im Gegensatz zu SML, was eine strikt auswertende Umsetzung von ML ist.

GHC Der Glasgow Haskell Compiler hält sich ebenso wie der HBC an den Haskell-Report. Im GHC gibt es Erweiterungen, die die Darstellung von Ein- und Ausgabe in funktionalen Programmen betreffen. Der GHC ist vollständig in Haskell geschrieben und erzeugt, anders als LML oder HBC, C-Code als Zwischencode, der dann von einem auf dem System verfügbaren C-Compiler in ein ausführbare Programm übersetzt wird. Diese Tatsache macht den GHC äußert portierbar und wir haben ihn auch auf den HPs des KI-Netzes zum Laufen gebracht.

[1] Miranda ist ein Warenzeichen von Research Software Limited

Gofer Compiler Zu Gofer gibt es einen Compiler, der die in Gofer geschriebenen Programme nach C compiliert. Das Kommando `gofc` ruft ihn auf. Es kann bei größeren Programmen zu C-Funktionen kommen, die der C-Compiler nicht mehr übersetzen kann. Außerdem fehlen viele (fast alle) der optimierenden Programmtransformationen, die in den anderen Compilern verwendet werden.

Clean Clean [NSvP91] ist eine nicht-strikte funktionale Sprache, die an der Universität in Nimwegen entwickelt wurde. Sie ist für Suns, OS2 und Macs [2] implementiert. Weitere Portierungen sind in Arbeit. Sie ist ursprünglich als Zwischensprache für einen Kompilierer aus einer höheren funktionalen Sprache (Miranda) entwickelt worden, hat aber mit Version 1.0 den Stand von Haskell errreicht (was Sprachsyntax und Ausdrucksmächtigkeit betrifft). Haskellprogramme lassen sich somit schnell ein-zu-eins in Clean-Programme übersetzen. Besonders ist an Clean das I/O Konzept, das es erlaubt, alle möglichen Fenster-, Maus- etc Anwendungen zu programmieren. Die I/O-Module sind dabei für alle Systeme implementiert. Man hat also mit einem Programmtext Anwenderprogramme für verschiedenste Plattformen.

Eine Striktheitsanalyse [Nöc93] erlaubt effizienten Code zu generieren. Eine ganze Reihe beeindruckende Beispielprogramme sind dem Programmpaket beigelegt. (Darunter ein Spreadsheet, ein Zeichenprogramm, Tetris)

Yale-Haskell Yale-Haskell hält sich auch an den Haskell-Report. Es erzeugt ähnlich wie der GHC C-Code erzeugt Lisp. Yale-Haskell ist für verschiedene Plattformen portiert, darunter auch der Macintosh. Man braucht allerdings zusätzlich einen Lisp-Compiler, der aus dem erzeugten Lisp-Code ausführbare Programme erzeugt. Yale-Haskell ist nicht am RBI installiert.

2.4 Einführungsliteratur

Die meiste Literatur über lazy funktionales Programmieren bezieht sich auf Miranda. Man kann sich aber relativ einfach von der Syntax bei Miranda auf Haskell/Gofer einstellen und lernt dabei dann gleich noch mehr. Es gibt mitlerweile eine ganze Reihe von Einführungen. Ein gern benutztes Einstiegsbuch ist hierbei [BW88]. Ebenfalls auf Miranda bezieht sich [Hol91]. Ein auf [BW88] aufbauendes, jedoch für Gofer geschriebenes Skript ist [Cun94]. Ein Buch, das sich an Haskell orientiert ist [Dav92]. Hier findet sich auch ein bißchen mehr theoretische Hintergrund. Für anglophobe gibt es auch eine Einführung auf deutsch in [Hin92]. Ein etwas älteres Buch noch aus Lisp-Zeiten ist [Hen80]. Für den zweiten Teil der Vorlesung sind [PJL91] und in Teilen auch [PJ87] zu empfehlen. Eine Einführung aus der Clean-Gruppe ist [PvE93], die auch auf Miranda basiert. Eine Lehrbuch

[2] Die Version für Macintosh kann bei uns kopiert werden

für Clean scheint in Arbeit, zumindest gibt es die ersten Kapitel als Beigabe des Clean-Pakets.

3 Programmieren in Hugs/Gofer

3.1 literate programming

Es sind zwei Arten von Programmskripts möglich: Programme, in denen die Kommentarzeilen mit der Zeichenfolge `--` markiert sind, oder Programme, in denen Codezeilen mit `>` markiert sind. Die zweite Art eignet sich, wenn man zu einem Programm viel theoretische Erklärungen hat. Dann kann das Progrmm im LaTeX-Format geschrieben werden, in dem der Code in der Verbatim-Umgebung steht. Der Hugs-Interpreter wird durch die Kommandos `:set -l` bzw. `:set +l` von den einen in den anderen Modus umgestellt.

3.2 Worte der Warnung

Hugs und Haskell ersparen dem Benutzer, Blöcke durch entprechende Worte oder Klammern zu markieren und Zeilen mit einem entsprechenden Zeichen zu markieren. Das hat aber seinen Preis: Die Aufgabe der Strukturierung übernehmen jetzt Einrückungsregeln. Verletzt man diese kann es zu Fehlermeldungen kommen, die dann unverständlich erscheinen. Betrachten wir ein Beispiel, in dem wir zwei Funktionen definieren wollen, eine die aus einer Liste eine Liste macht in der nur noch jedes zweite Element der Liste enthalten ist und eine die das gleiche macht, aber für jedes dritte Element. Dazwischen Kommentar.

```
> zweiteElemente :: [a] -> [a]
> zweiteElemente [] = []
> zweiteElemente x:[] = []
> zweiteElemente _:x2:xs = x2:(zweiteElemente xs)

Das hier ist Kommentar, der nat"urlich viel l"anger sein k"onnte...

>  dritteElemente :: [a] -> [a]
>  dritteElemente [] = []
>  dritteElemente _:[] = []
>  dritteElemente _:_:[] = []
>  dritteElemente _:_:x3:xs = x3:(dritteElemente xs)
```

Das Programm enthält Fehler. Hugs meldet beim Einlesen:

```
Reading script file "usegofer.tex":
Parsing
ERROR "usegofer.tex" (line 73): Syntax error in input (unexpected `=')
```

Die 73. Zeile war in diesem Fall die Zeile für `dritteElemente []`. Die Fehlermeldung läßt einen hier also im Dunklen darüber, was wirklich falsch ist. Es ist die Einrückung. Da die Zeilen von `dritteElemente` ein Leerzeichen weiter eingerückt sind als die Zeilen von `zweiteElemente`, versucht Hugs diese als Fortsetzung der letzten Zeile von `zweiteElemente` zu interpretieren, was schief geht. *Es ist also wichtig auf Einrückung zu achten.* Besonders sollte man sich nicht durch Fehlermeldungen über falsche Semikolonzeichen verwirren lassen. Textuell wird das Programm durch Einrücken strukturiert; intern aber durch Semikolonzeichen. Die Fehlermeldungen beziehen sich manchmal auf diese.

3.3 Tips zur Formulierung

In vielen Fällen ist es so, daß es viele richtige Arten der Formulierung eines Programmteils gibt. Gibt es „bessere" oder „schlechtere" Formulierungen? Davie schreibt dazu in [Dav92], daß das „Ich kann das mit weniger Symbolen schreiben als Du" uns nicht recht weiterbringt und daß das „Ich kann das eleganter schreiben als Du" uns möglicherweise mehr Einsicht in die Arbeitsweise des Algorithmus liefert, aber eine Freizeitbeschäftigung ist auf die man beliebig viel Zeit verschwenden kann. Nach Ihm sollten wir nach dem „Ich kann das in geringerer Zeit schreiben als Du" streben.
Meiner Meinung nach ist das nur die halbe Wahrheit, denn eine übersichtliche (meistens auch kurze) Funktion und eine leicht verständliche (meist auch elegante) Formulierung ermöglichen erst ein Programm in kurzer Zeit zu schreiben, denn man wird es bis es fertig ist oft lesen, verstehen und darüber nachdenken müssen.
Die Erfahrung zeigt, daß eine Funktion häufig dadurch leichter zu implementieren ist, daß man sich überlegt, aus welcher allgemeineren Funktion sie sich durch Spezialisierung ergibt. Fast immer gewinnt eine Funktion an Verständlichkeit, wenn statt `map` und `filter` list comprehensions verwendet werden. Die Funktionen benötigen dann weniger Platz und die Mengenschreibweise ist einem im allg. vertrauter als die Formulierung mit Hilfe von `map` und `filter`.
Es ist anzuraten, eher mehrere kleinere Funktionen aus nur ein paar Zeilen zu definieren, als umfangreiche Funktionen, die sehr komplexe Probleme lösen. Man kann im Interpretierer jede dieser kleinen Funktionen schnell durchtesten, was die Fehlersuche vereinfacht.

3.4 Wahl des Editors

Vom RBI werden mehrere Editoren angeboten: u. a. `vi`, `me`, `TextEdit`, `emacs` etc. Welcher ist für die Arbeit an Haskell- oder Hugs-Programmen am geeignetsten? Kann man diese Frage so einfach beantworten? Nein! Das hängt sicher vom persönlichen Geschmack ab, trotzdem möchte ich erklären warum meines Erachtens der `emacs` der am Besten geeignete ist.

- Hugs gibt seine Fehlermeldungen mit Zeilennummern aus, also ist ein Editor vorteilhaft, der eine beliebige Zeile des Programms direkt anspringen kann. `me`, `TextEdit` und `emacs` können das, bei `vi` weiß ich keine andere Art als an den Anfang des Programms zu gehen und von dort aus die entsprechende Anzahl an Zeilen vorwärts.

- emacs hat einen Lisp-mode, der vor allen die Klammern richtig zählt.

- Der Funktionsumfang von `TextEdit` ist sehr beschränkt.

- `me` ist dem `emacs` recht ähnlich, aber lange nicht so leistungsfähig, wie `emacs`. (Für die, die es noch nicht wissen `me` steht für *micro emacs*). Das zweimalige Drücken der ESCAPE- (bzw. META) -Taste in `me` läßt ein kleines Pulldown-Menue erscheinen, durch das mit den Cursortasten navigiert werden kann. Für den Einstieg ist dieser Editor eine bequeme Sache.

- Sicherlich erfordert `emacs` etwas Gewöhnung, möglicherweise etwas mehr als die anderen Editoren, aber da `emacs` alles erfüllt, was man sich von einem Editor wünscht [3] zahlt sich dies langfristig aus.

- emacs ist übrigens in Lisp geschrieben. Also ein Beispiel für ein großes in einer funktionalen Sprache enwickeltes Programm.

Im Hugs-Interpretierer gibt es das `:edit`-Kommando, das direkt den für Hugs als Default-Editor angegebenen Editor mit dem aktuellen Programm aufruft. Hier ist aber leider auch eine kleine Falle versteckt: Nach einer Fehlermeldung wird nicht mehr der Default-Editor aufgerufen, sondern `vi` (da irgendwelche Schnittstelleneigenschaften verlangt werden, die `me` nicht bereithält. Dieses kann verwirrend bis nervig sein, da immer mal wieder ein anderer Editor erscheint. Deshalb ist es ratsam, das Programm in einem eigenen Fenster zu edieren und nicht das `:edit`-Kommando in Hugs zu verwenden.

[3] `emacs` kann sogar noch etwas mehr als man sich von einem Editor wünscht, z. B.: man kann `emacs` als Filemanager verwenden, mit ihm Dateien von einem FTP-Server holen, die USENET News lesen/schreiben, EMail lesen/schreiben, aus ihm heraus TeX oder einen Previewer starten, Lisp Programme schreiben und laufen lassen und vieles mehr.

4 Haskell-Kurzübersicht

4.1 Funktionen und Operatoren

4.1.1 Funktionsnamen

Funktionen werden in Haskell dargestellt, entweder durch einen Namen (z.B.: fac, qsort) oder durch einen Operator (z.B.: *, +). Funktionsnamen beginnen mit einem kleinen Buchstaben gefolgt von einer beliebigen Anzahl von Buchstaben, Ziffern, dem Apostroph (') und dem Unterstrich (_). Beispiele für erlaubte Funktionsnamen sind:

sum integerSum f" do'until'zero

Folgende, in Haskell reservierten Wörter, dürfen nicht als Funktionsnamen verwendet werden:

case	of	where	let
in	if	then	else
data	type	infix	infixl
infixr	primitive	class	instance

Operatorsymbole werden durch ein oder mehrere der folgenden Zeichen dargestellt:

`: ! # $ % & * + . / < = > ? @ \ ^ | - ~ \`

(~ nur an der ersten Position, `--` darf nicht am Anfang stehen (s. Kommentare))
Operatoren mit spezieller Bedeutung in Haskell sind:

`:: = .. @ \ | <- -> ~ =>`

4.1.2 Infix-Darstellung von Funktionen

Durch das Einschließen von Funktionsnamen in Apostrophe (Achtung: 'backquotes', ALT 96 auf dem PC) können Funktionsnamen genauso wie Operatoren verwendet werden:

5 ‘plus‘ 6

Die Darstellung x ‘op‘ y ist äquivalent zu der Präfixdarstellung op x y.

4.1.3 Präfixdarstellung von Operatoren

Durch das Einschließen eines Operators in Klammern kann dieser wie Funktionen in Präfixdarstellung benutzt werden: (+) 5 6 ist äquivalent zu 5 + 6.

4.1.4 Vorrang und Assoziativität von Operatoren

Die Assoziativität eines Operators bestimmt die Reihenfolge, in der ein Ausdruck bewertet wird. Die Subtraktion beispielsweise ist linksassoziativ, d.h $1 - 2 - 3$ wird wie $(1 - 2) - 3$ ausgewertet, während das Potenzieren rechtsassoziativ, also `a^b^c` äquivalent zu `a^(b^c)` ist.
Vorrang und Assoziativität der Operatoren in Haskell (definiert in der Standard-Prelude):

Operator	Vorrang	Assoziativität
`!!`	9	linksassoziativ
`.`	9	rechtsassoziativ
`^`	8	rechtsassoziativ
`*`	7	linksassoziativ
`/ ‘div‘ ‘rem‘ ‘mod‘`	7	
`+ -`	6	linksassoziativ
\\\\	5	
`++ :`	5	rechtsassoziativ
`== /= < <= > >=`	4	
`‘elem‘ ‘notElem‘`	4	
`& &`	3	rechtsassoziativ
`||`	2	rechtsassoziativ

Standardmäßig werden Operatoren in Haskell als nicht–assoziativ und mit Vorrang 9 behandelt. Um dies zu ändern stehen folgende Befehle zur Verfügung:

infixl *vrrg op*	Operator *op* ist linksassoziativ und hat Vorrang *vrrg*
infixr *vrrg op*	Operator *op* ist rechtsassoziativ und hat Vorrang *vrrg*
infix *vrrg op*	Operator *op* ist nicht-assoziativ und hat Vorrang *vrrg*

Beispiele:

```
infixl 7 *
infixr 8 ^
infix 7 /
```

4.2 Typen in Haskell

4.2.1 Typ–Namen

Typ–Namen in Haskell beginnen mit einem großen Buchstaben gefolgt von einer beliebigen Anzahl von Buchstaben, Ziffern, dem Apostroph (’) und dem Unterstrich (_).

4.2.2 Der Typ-Operator (::)

Durch den Operator :: kann beliebigen Ausdrücken in Haskell ein Typ zugeordnet werden:

42 :: Int   42 ist ein Integer-Wert

4.2.3 Der Funktions–Operator (->)

Der Typ von Funktionen wird mit Hilfe des 'Funktions– Operators' dargestellt:

`fac :: Int -> Int`	`fac` ist eine Funktion, die einen Integer–Wert als Argument erwartet und einen Integer–Wert als Ergebnis liefert

4.2.4 Standard-Typen in Haskell

Bool	Ein Ausdruck vom Typ Bool hat den Wert True oder False
Int	Eine ganze Zahl
Float	Eine Gleitkomma-Zahl
Char	Ein Zeichen
[...]	Eine Liste; [Int] steht für eine Liste ganzer Zahlen; [Char] für eine Zeichenkette
(...,...)	Tupel; (Int,Int) steht für ein Paar aus ganzen Zahlen; (Int, [Char], Bool) steht für ein Tripel bestehend aus einer ganzen Zahl, einem String und einem Boole'schen Wert; (5, "Gofer", True) :: (Int, [Char], Bool)

4.2.5 Listen

Listen werden durch eckige Klammern dargestellt: [Int] ist der Typ einer Liste mit ganzen Zahlen. Die leere Liste wird durch [ ] dargestellt. Alle Elemente einer Liste müssen den gleichen Typ besitzen.

4.2.6 Zeichen

Zeichenkonstanten werden durch Einschließen in Apostrophe dargestellt:

'a' :: Char

4.2.7 Zeichenketten

Zeichenketten werden in Haskell als Listen von einzelnen Zeichen, also [Char], dargestellt. Die Zuweisung einer Zeichenkette erfolgt durch Einschließen in doppelte Anführungsstriche:

```
a :: [Char]
a = "Gofer"
```

4.2.8 Escape-Sequenzen, Steuerzeichen

Um Steuerzeichen in eine Zeichenkette einzufügen, wird ein Backslash (\) gefolgt von einem oder mehreren Zeichen benutzt:

\\	Backslash
\'	Apostroph
\n	neue Zeile
\b oder \BS	Backspace
\del	Delete
\t oder \HT	Tabulator
\a oder \BEL	Bell
\f oder \FF	Formfeed
\nummer	Zeichen mit ASCII-Wert nummer
\onummer	Zeichen mit ASCII-Wert nummer (oktale Darstellung)
\xnummer	Zeichen mit ASCII-Wert nummer (hexadezimale Darstellung)

4.2.9 Polymorphismus

In Haskell ist es möglich polymorphe Funktionen zu definieren, also Funktionen, die nicht auf einen bestimmten Argument- und Ergebnistyp festgelegt sind. Ein Beispiel hierfür ist die Funktion map, die eine Funktion und eine Liste als Argumente erwartet und eine Liste liefert, die durch die Anwendung der Funktion auf die einzelnen Elemente der Liste entsteht. Map arbeitet auf Listen beliebigen Typs. Die Definition solcher polymorpher Funktionen erfolgt mit Typvariablen:

```
map :: (a -> b)    ->   [ a ]     ->   [ b ]
        erstes         zweites      Ergebnistyp
       Argument       Argument       ist Liste
     ist Funktion    ist Liste
```

4.2.10 Benutzerdefinierte Datentypen

Benutzerdefinierte Datentypen werden mit dem `data`–Befehl deklariert:

```
data Datatype a1 ... an = constr1 | ... | constrm
```

wobei `Datatype` der Name (beginnend mit einem großen Buchstaben) des neuen n-stelligen Typkonstruktors ist ($n \geq 0$), $a1 \ldots an$ verschiedene Typvariablen sind und `constr1` ...`constrm` ($m \geq 1$) die Art beschreiben, in der die Elemente des neuen Datentyps konstruiert werden. Dabei können die Konstruktoren (`constr`) als Konstruktorfunktionen (beginnend mit einem großen Buchstaben) oder als Konstruktoroperatoren (beginnend mit einem Doppelpunkt) dargestellt werden:

```
data Tag = Son | Mon | Die | Mit | Don | Fre  |  Sam
```

neuer 0-stelliger Datentyp Tag, der als Wert eine von sieben 0-stelligen Konstruktorfunktionen annehmen kann

```
data Temp = Celsius Float | Fahrenheit Float
a :: Temp
a = Celsius 0.0
a = Fahrenheit 32.0
```

0-stelliger Datentyp Temp mit zwei einstelligen Konstruktorfunktionen

```
data Set a = Set [a]
m :: Set Int
m = Set [1, 2, 3]
n :: Set Float
n = [1.4, 4.8]
```

einstelliger Datentyp mit einstelliger Konstruktorfunktion

```
data Tree a = Lf a | Tree a :^: Tree a
b :: Tree Int
b = Lf 2 :^: ( Lf 4 :^: Lf 6)
```

einstelliger Datentyp mit Konstruktorfunktion und Konstruktoroperator

4.2.11 Typ-Synonyme

Typ-Synonyme gestatten die Definition von Abkürzungen für beliebige Ausdrücke:

```
type Name a1 ... an = expansion
```

`Name` muß mit einem großen Buchstaben beginnen und `a1 ...an` verschiedene Typvariablen sein. Bsp.:

```
type String = [Char]
```

4.3 Funktionsdefinition

4.3.1 Pattern Matching

Funktionen werden in Haskell definiert, durch eine oder mehrere Gleichungen der Form:

```
f pat1 pat2 ... patn = rhs
```

(oder durch einen äquivalenten Ausdruck, bei dem die Funktion durch einen Operator dargestellt wird — s. Funktionsnamen). Jeder der Ausdrücke `pat1 pat2 ... patn` representiert ein Argument der Funktion `f` und wird als 'Pattern' bezeichnet. Die Anzahl der Pattern bestimmt also die Stelligkeit der Funktion. Wird `f` durch mehr als eine Gleichung definiert, so müssen alle Gleichungen hintereinander stehen und die gleiche Stelligkeit besitzen.
Bei der Ausführung einer Funktion, die durch mehr als eine Gleichung definiert wurde, findet ein Pattern–Matching statt, bei dem die Argumente der Funktion mit den Pattern der Definitionsgleichungen verglichen werden (in der Reihenfolge, in der sie definiert wurden). Die erste Gleichung, bei der die Pattern mit den Argumenten der Funktion übereinstimmen, wird ausgeführt:

```
hello "Mark" = "Howdy"
hello name   = "Hello " ++ name ++ ", nice to  meet you!"
```

Aufruf: hello "Mark" ergibt: `Howdy`
Aufruf: hello "Fred" ergibt: `Hello Fred, nice to meet you!`

```
hello name = "Hello " ++ name ++ ", nice to  meet you!"
hello "Mark" = "Howdy"
```

Aufruf: hello "Fred" ergibt: `Hello Fred, nice to meet you!`
Aufruf: hello "Mark" ergibt: `Hello Mark, nice to meet you!`

4.3.2 Pattern

Außer Variablen und Konstanten (s. o.) sind als Pattern erlaubt:

_	Wildcard, stimmt (wie eine Variable) mit jedem Wert überein
otherwise	äquivalent mit _
(x,y)	Tupel, stimmt überein mit einem Paar; auf des erste Element des Paares kann mit der Variablen x, auf das zweite mit y zugegriffen werden
[x]	bezeichnet eine Liste mit genau einem Element, auf das mit der Variablen x zugegriffen werden kann
[_,2,_]	bezeichnet eine Liste mit genau 3 Elementen, wobei das zweite Element gleich 2 sein muß
[ ]	bezeichnet die leere Liste
(x:xs)	bezeichnet eine nichtleere Liste mit head x und tail xs
(x:y:ys)	bezeichnet eine Liste mit mindestens zwei Elementen
p@(x,y)	bezeichnet ein Paar; gestattet zusätzlich den Zugriff auf das gesamte Paar mit dem Namen p (und nicht nur auf die einzelnen Elemente mit x und y)
(m+1)	bezeichnet einen Integer-Wert, der größer oder gleich 1 ist

4.3.3 Guarded Equations

Jede Gleichung in einer Funktionsdefinition kann 'guards' enthalten, die Bedingungen angeben, welche von den Argumenten der Funktion erfüllt werden müssen:

```
f x1 x 2 ... x1 | bedingung1 = e1
                | bedingung2 = e2
                ...
                | bedingungm = em
```

Beispiel:

```
oddity n | even n = "even"
         | otherwise = "odd"
```

4.3.4 Lokale Definitionen

Lokale Definitionen können in Haskell entweder durch einen let-Ausdruck oder durch eine where-Klausel dargestellt werden:

```
let decls in expr
```

Beispiel:

```
f = let
        a = 4
        b = 5
    in a + b
```

Funktionsdefinition mit `where`:

```
f = a + b
    where
         a = 5
         b = 6
```

(Achtung: das Schlüsselwort `where` muß gegenüber dem Funktionsnamen `f` eingerückt werden (s. Layout-Regeln))

4.3.5 Unendliche Datenstrukturen

Lazy Evaluation erlaubt es in Haskell unendliche Datenstrukturen darzustellen. Z. B. könnte die unendliche Liste der Quadratzahlen aller natürlichen Zahlen wie folgt dargestellt werden:

```
powersofTwo = 1 : map double powersofTwo
       where double n = 2 * n
```

4.3.6 Funktionen höherer Ordnung

In Haskell können Funktionen genauso verwendet werden wie andere Werte (z.B. Integer). Sie können als Argumente an andere Funktionen übergeben werden und das Ergebnis anderer Funktionen sein. Die Typdefinition solcher Funktionen erfolgt mit Typvariablen.
Beispiel: Der .–Operator (Funktionskomposition):

```
        1. Argument    2. Argument    Ergebnistyp
(.) ::    (b -> c)  ->   (a -> b)   ->  (a -> c)
(f.g) x = f (g x)
```

4.4 Spezielle Sprachkonstrukte

4.4.1 Arithmetische Folgen

Haskell gestattet die einfache Erzeugung von arithmetischen Folgen (Listen):

[n ...]	stellt die unendliche Liste dar, beginnend mit n, in aufsteigender Reihenfolge. Bsp.: [2 ...] = [2, 3, 4, 5, etc ...
[n ... m]	stellt die Liste der Elemente von n bis m in aufsteigender Reihenfolge dar: [-3 ... 1] = [-3, -2, -1, 0, 1]; [9 ... 0] = []
[n, m ...]	stellt die Liste dar, die mit n und m beginnt. Die folgenden Elemente werden durch die Differenz aus m und n bestimmt: [1, 3, ...] = [1, 3, 5, 7, etc ...; [2, 1, ...] = [2, 1, 0, -1, -2, etc ...; [0, 0, ...] = [0, 0, 0, 0, 0, etc ...
[n, n' ... m]	stellt die Liste von n bis m dar, wobei die Schrittweite durch die Differenz aus n' und n bestimmt wird: [1, 3 ... 12] = [1, 3, 5, 7, 9, 11]; [0, 0 ... 10] = [0, 0, 0, 0, etc ...

4.4.2 List Comprehensions

List comprehension ist eine sehr leistungsfähige Technik zur Listen-Verarbeitung. Die Grundstruktur von list comprehension ist:

```
[ expr | qualifiers ]
```

wobei `qualifiers` eine der folgenden drei Formen annehmen kann:

- Generatoren: Ein `qualifier` der Form `pat <- expr` wird benutzt, um aus der Liste `expr` alle Elemente zu extrahieren, die dem Pattern `pat` genügen (in der Reihenfolge in der sie in `expr` auftreten). Bsp.:
  `[x*x | x <- [1 .. 10]] = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]`
- Filter: Ein Boole'scher Ausdruck als `qualifier` dient als Filter:
  `[ e | condition ] = if condition then [e] else []`
- Lokale Definition: Ein `qualifier` der Form `pat = expr` dient zur Definition einer lokalen Variablen. Es gilt:
  `[ e | pat = expr ] = [ let pat = expr in e ]`

Innerhalb einer list comprehension können mehrere `qualifiers` angegeben werden, die jeweils durch ein Komma getrennt werden müssen.
Beispiele:

```
[ (x,y) | x <- [1 .. 3], y <- [1 .. 2] ] =
        [(1,1), (1,2), (2,1), (2,2), (3,1), (3,2)]

[ (x,y) | x <- [1 .. 3], y <- [1 .. x] ] =
        [(1,1), (2,1), (2,2), (3,1), (3,2), (3,3)]

[ x | x <- [1 .. 10] ] = [ 2, 4, 6, 8, 10 ]

[ x | x <- [1,2], x <- [3, 4] ] = [ 3, 4, 3, 4]

[ x | x <- [ [1, 2], [3, 4] ], x <- x ] = [1, 2, 3, 4]
```

4.4.3 Lambda Ausdrücke

Lambda Ausdrücke gestatten die Definition namenloser Funktionen:

\ *atomicPatterns* `->` *expr*

Beispiele für Lambda Ausdrücke:

```
(\x -> x * x)
(\x y -> x + y)
```

4.4.4 Case Anweisung

Die case Anweisung dient zur Auswertung eines Ausdrucks und der Auswahl (in Abhängigkeit des Ergebnisses) einer von (mglw.) mehreren Alternativen. Die Syntax lautet:

`case` *exp* `of`
    pat1 `->` *exp1*
    pat2 `->` *exp2*
    ...
    patn `->` *expn*

Beispiele:

```
case list of
        []        -> "leere Liste"
        otherwise -> "Liste nicht leer"
```

Trifft keine Alternative auf den ausgewerteten Ausdruck zu, erfolgt eine Fehlermeldung. Auf der rechten Seite einer Alternativen (*pat* `->` *exp*) können zusätzlich Guards (s.a. Guarded Equations) verwendet werden:

`case` *exp* `of`
    pat1 `|` *condition1* `->` *exp1*

condition2 -> *exp2*
...
patn | *conditionn1* -> *expn1*
conditionn2 -> *expn2*

4.4.5 Irrefutable (lazy) Pattern

Das Voranstellen einer Tilde (˜) vor ein Pattern hat zur Folge, daß das Pattern nicht zu Beginn einer Funktion überprüft wird, sondern erst zu dem Zeitpunkt, zu dem es tatsächlich benötigt wird. Dies gestattet es ein Pattern zu erzeugen, das zu jedem Ausdruck paßt (z.B. ein undefinierter Ausdruck oder eine noch nicht vorhandene Response des Betriebssystems, vgl. Ein- und Ausgabe):

`(\(x, y) -> 0) undefined = { undefined}`
Fehlermeldung, da 'undefined' nicht die Form (x, y) hat. ('undefined' ist ein Ausdruck, der bei seiner Auswertung zu einem 'run-time error' führt)
`( ˜ (x, y) -> 0) undefined = 0`
Bei der Ausführung wird nicht überprüft, ob das Argument (undefined) die Form des Pattern hat. Da der Rumpf der Funktion das Argument nicht benötigt, liefert die Funktion ein Ergebnis.

4.4.6 Kommentare

Es gibt zwei Möglichkeiten in Haskell Kommentare einzufügen:

- Ein einzeiliger Kommentar beginnt mit zwei Minuszeichen (−−) und endet am Zeilenende (aus diesem Grund kann kein selbstdefiniertes Operatorsymbol mit −− beginnen)

- Ein Kommentar über (mglw.) mehrere Zeilen wird mit {− eingeleitet und mit −} beendet. Auf diese Weise geklammerte Kommentare können in andere Kommentare eingebettet werden, es ist also auf die korrekte Klammerung zu achten:
  {− Dies {− ist ein −} korrekter {− Kommentar −} −}
  {− Dies führt zu einer {− Fehlermeldung −}

4.5 Layout rules - Einrücken von Programmcode

Haskell benutzt das Einrücken einzelner Programmteile gegenüber dem Rest des Programms zur Erkennung der Struktur des Programms (Geltungsbereiche von lokalen Definitionen (where, let) und Alternativen eines case Ausdrucks).
Beispiel:

```
f x y = g ( x + w )
        where g u = u + v
                where v = u * u
        w = 2 + y
```

Durch das Einrücken der zweiten 'where'-Klausel wird deutlich, daß `v` eine lokale Variable der Definition von `g` ist, während `w` (genauso weit eingerückt wie die Definition von `g`) zu der Definition von `f` gehört.
Eigentlich erwartet Haskell ein Programm, bei dem die Struktur durch Klammerung mit '{ ' und '} ' (bei mehreren, getrennt durch ';') eindeutig wird:

```
f x y = g ( x + w )
        where { g u = u + v
                where { v = u * u
                }; w = 2 + y
        }
```

Die Eingabe dieser Klammerung ist jedoch nicht nötig, da sie durch das Einrücken einzelner Programmteile automatisch eingefügt werden kann. Dabei wird nach folgenden Regeln (Layout-Rules) verfahren:

- Eine öffnende Klammer ('{ ') wird am Beginn des Programms und nach jedem der Schlüsselworte where, let oder of eingefügt.
- Ein Semikolon wird vor jedem Token eingefügt, das genauso weit eingerückt ist, wie das Token nach der letzten öffnenden Klammer.
- Eine schließende Klammer wird vor einem Token eingefügt, das weniger weit eingerückt ist, als das Token nach der letzten öffnenden Klammer oder wenn ein 'unerwartetes' Token (z.B. ein Schlüsselwort) auftaucht. Dies laßt es zu, daß z.B. folgender Code korrekt geklammert wird:

```
let a = fact 12 in a + a
```

wird übersetzt in

```
let {a = fact 12} in a + a
```

('in' ist unerwartetes Token)
Leere Zeilen (mit tabs und blanks) und Kommentare haben keinen Einfluß auf die Layout-Rules. Tabs werden als 8 Zeichen Einrücktiefe betrachtet und das EOF-Token muß Einrücktiefe Null besitzen (d.h. hinter der letzten Zeile des Files, die Text enthält (auch Kommentare) muß (mind.) ein Return folgen).

4.6 Ein- und Ausgabe in Haskell

Von einem Programm, das über I/O Operationen mit dem Betriebssystem kommuniziert haben wir in der Regel die folgende Vorstellung:

```
(print stdout "Geben sie eine Zahl ein");
(read stdin number);
(print stdout "das Quadrat von ");
(print stdout (int2string number));
(print stdout "ist: ");
(print stdout (int2string (square number)))
```

Von derartigen Programmen interessiert nicht mehr der Wert einer Funktion sondern ihr Seiteneffekt, in Form eines Bildschirmausdrucks oder einer geänderten Datei. Auch Haskell stellt solche Programme zur Verfügung. Sie haben den Typ:`IO()`. Wenn eine I/O-Operation nicht nur einen Seiteneffekt hat, sondern auch ein Ergebnis liefert (z.B. den Dateiinhalt nach lesen einer Datei, so ist der Typ dieser I/O-Operation: `IO` *Typ*, wobei *Typ* der Ergebnistyp ist. So gibt es z.B. die I/O-Funktionen:

```
putChar      :: Char -> IO ()
putStr       :: String -> IO ()
putStrLn     :: String -> IO ()  -- adds a newline
print        :: Show a => a -> IO ()

getChar       :: IO Char
getLine       :: IO String
getContents   :: IO String
interact      :: (String -> String) -> IO ()

type FilePath =  String

writeFile     ::  FilePath -> String           -> IO ()
appendFile    ::  FilePath -> String           -> IO ()
readFile      ::  FilePath                     -> IO String
```

Reines funktionales Programmieren ist nun aber auf das Programmieren ohne Seiteneffekte ausgelegt. Funktionen, die Seiteneffekte haben, müssen in einer definierten Sequenz abgearbeitet werden. In funktionalen Programmen kann man aber nur schwer so programmieren, daß eine Auswertungsreihenfolge von Funktionsaufrufen deutlich sichtbar ist. Es wurden nun im Prelude Funktionsoperatoren definiert, die genau eine solche Sequenzierung erlauben. Mit ihnen kann man zwei I/O-Operationen verknüpfen; wobei die eine für I/O-Operationen ohne Ergebnis und die andere für I/O, das ein Ergebnis liegert, zuständig ist. Die Sequenzoperatoren sind in Infix-Schreibweise definiert:

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

Das obige Programm läßt sich jetzt einfach imperativ schreiben als:

```
main =
 putStr "Geben sie eine Zahl ein "          >>
 getLine                                    >>= \number ->
 putStr "das Quadrat von "                  >>
 print ((read number)::Int)                 >>
 putStr "ist: "                             >>
 print ((read number)*(read number))
```

Wie wir sehen übernehmen die Operatoren die Sequenzierung, wie es in imperativen Sprachen zumeist das Semikolon macht. Für eine Folge von I/O-Programmen die per Sequenz verkettet sind, gibt es noch eine gezuckerte Schreibweise, die unser obiges Programm in folgender Form darstellt:

```
main = do {putStr "Geben sie eine Zahl ein ";
           number <- getLine;
           putStr "das Quadrat von ";
           print ((read number)::Int);
           putStr "ist: ";
           print ((read number)*(read number))}
```

Dieses ist die sogenannte *do*-Schreibweise. Hier kommt das gewohnte Semikolon als Trennung einer Sequenz von Befehlen wieder zu Ehren.

4.7 Überladen in Haskell

Überladen gestattet die Definiton von Funktionen (bzw. Operatoren), deren Bedeutung vom Typ ihrer Argumente abhängig ist. So kann z.B. der Gleichheitsoperator (==) sowohl auf Integer, als auch auf Characters (usw.) angewendet werden.
Alle Typen, für die ein (überladener) Operator existiert, werden in Typklassen zusammengefaßt. Die Klasse Eq enthält z.B. alle Typen, die auf Gleichheit getestet werden können.
Eine Typklasse besteht aus drei Teilen:

- einem Namen (z.B. Eq)
- Funktionen (member functions), die überladen sind (z.B. (==) und (/=) bei der Klasse Eq)

- Typen (instances), auf denen die member functions definiert sind (bei Eq: Int, Float, Char, [Int], (Int, Int), ...)

Die Definition einer Typklasse erfolgt mit dem Schlüsselwort `class`:

```
class Eq a where
        (==), (/=) :: a -> a -> Bool
        x /= y  =  not (x == y)
```

definiert eine neue Klasse Eq, die die member functions (==) und (/=) enthält und die Definition des Operators (/=) mit Hilfe der Operators (==) angibt. Der Typ dieser Operatoren lautet jetzt:

```
Eq => a -> a -> Bool
```

Ein Typ, auf dem die Operatoren einer Typklasse arbeiten sollen, muß dieser Typklasse zugeordnet werden. Dies geschied mit Hilfe des Schlüsselwortes `instance`:

```
instance Eq Bool where
        True == True = True
        False == False = True
        _ == _ = False
```

fügt der Typklasse Eq den Typ Bool hinzu, und definiert den Operator (==) auf diesem Typ. Der Operator (/=) muß nicht definiert werden, da er bereits bei der Klassendefinition mit Hilfe des Gleichheitsoperators definiert wurde. Nach dieser Definition können also Bool'sche Werte auf Gleichheit überprüft werden.
Weitere Beispiele:

```
instance Eq Char where
        c == d   =   ord c = ord d
```

Gestattet Zeichen auf Gleichheit zu testen. Zwei Zeichen sind gleich, falls ihre ASCII– Codes gleich sind.

```
instance  Eq  a => Eq [a] where
        [] == []         =  True
        [] == (y:ys)     =  False
        (x:xs) == []     =  False
        (x:xs) == (y:ys) =  x == y && xs == ys
```

Bedeutung von (=>): Falls Typ a der Klasse Eq angehört, so soll auch der Typ [a] der Klasse Eq angehören. Anschließend muß der Operator (==) auf [a] definiert werden.

```
instance (Eq a, Eq b) => Eq (a,b) where
        (x, y) == (u, v) =  x == y && u == v
```

wenn Typ a und Typ b zu Eq gehören, sollen auch Paare aus Elementen vom Typ a bzw. b auf Gleichheit prüfbar sein (d.h. zur Klasse Eq gehören)

Literatur

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice-Hall International, London, 1988.

[Cun94] H. Conrad Cunningham. Notes on functional programming with gofer. Technical Report UMCIS-1994-01, Dept of Computer and Information Science, University of Mississippi, January 1994.

[Dav92] A.J.T. Davie. *An Introduction to Functional Programming Systems Using Haskell.* Cambridge University Press, Cambridge, 1992.

[Hen80] Peter Henderson. *Functional Programming: Application and Implementation.* Prentice-Hall International, London, 1980.

[Hin92] Ralf Hinze. *Einführung in die funktionale Programmierung mit Miranda.* Teubner, Stuttgart, 1992.

[Hol91] Ian Holyer. *Functional Programming with Miranda.* Pitman, 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* IT Press, Cambridge, Massachusetts, 1990.

[Nöc93] Eric Nöcker. Strictness analysis using abstract reduction. In *Functional Programming Languages and Computer Architecture*, pages 255–265. ACM Press, 1993.

[NSvP91] E. G. J. M. H. Nöcker, J. E. W. Smetsers, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Concurrent Clean. In Springer Verlag, editor, *Proc of Parallel Architecture and Languages Europe (PARLE'91)*, number 505 in Lecture Notes in Computer Science, pages 202–219, 1991.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International, London, 1987.

[PJL91] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages: a Tutorial.* Prentice-Hall International, London, 1991.

[PvE93] Rinus Plasmeijer and Marko van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, Workingham, 1993.