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

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

Praktikum: Praktische Informatik

Sommersemester 1998

Philipp de Graaff   Christian Mönch   Manfred Schmidt-Schauß
Michael Schneider   Marko Schütz   Uwe Zinndorf

20. April 1998

Inhaltsverzeichnis

1 Organisation

1.1 Sprechstunden

Während des Semesters werden die Tutoren zu festen Zeiten in den Fischer-Räumen Sprechstunden haben, zu denen sie bei Fragen oder Schwierigkeiten zur Verfügung stehen. Jeder Tutor betreut dabei eine Aufgabe.

Tutor	Aufgabe	Sprechzeit
Philipp de Graaff	3	Mo 14–18 Uhr
Michael Schneider	1	Mi 14–18 Uhr
Uwe Zinndorf	2	Do 14–18 Uhr

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

Philipp de Graaff	graaff@informatik.uni-frankfurt.de
Michael Schneider	m_schnei@informatik.uni-frankfurt.de
Marko Schütz	marko@informatik.uni-frankfurt.de
Uwe Zinndorf	zinndorf@informatik.uni-frankfurt.de

In weiteren Notfällen ist Marko Schütz in Raum 309 erreichbar.

1.2 Computerraum

Ebenfalls zu den Sprechzeiten, wird der Raum ohne Nummer gegenüber von Raum 21 bei den Fischer-Räumen für Praktikanten reserviert sein. Zusätzlich stehen natürlich auch die Rechner in den Fischer-Räumen zur Verfügung, so daß es zu keinen Rechnerengpässen kommen sollte.

1.3 Scheinvergabe

Die Scheine für des Praktikum werden für die korrekte und vollständige Bearbeitung der Aufgaben im folgenden Sinne vergeben: Es ist uns lieber, jemand fällt durch viele dumme bis kluge Fragen auf und hat schließlich ein nicht ganz so perfektes Programm, als jemand zeigt sich nur alle 3 Wochen und führt dann ein all-dancing-all-singing Programm vor. Also: den Schein gibt es für Leute, von denen man merkt, daß sie etwas lernen wollen. Haben wir von jemandem nicht diesen Eindruck werden wir ihn, sofern er sich mal zeigt, früh genug davon in Kenntnis setzen, so daß er etwas gegen diesen Eindruck unternehmen kann.

Es sollen je zwei Praktikanten für die Bearbeitung einer Aufgabe eine Gruppe bilden. Es ist nicht nötig, daß die Praktikanten alle Aufgaben in dieser Konstellation bearbeiten. Jeder Tutor wird für Fragen, die Präsentation der Arbeiten, die Abgabe der Ausarbeitungen etc. 4 Stunden in der Woche zur Verfügung stehen. Die jeweils erste Stunde davon soll zur Besprechung dienen und Anwesenheit ist daher während dieser ersten Stunde erforderlich.

Zur Erfüllung der Aufgaben gehört die Abgabe und Präsentation eines hier auf den *Rechnern des RBI* laufenden, kommentierten Programms, sowie eine schriftliche Ausarbeitung, die allerhöchstens 15 Seiten lang sein sollte, in der sowohl die Grundlagen, die zur Lösung der Aufgabe notwendig waren dargelegt werden, die Lösung beschrieben wird als auch der Prozeß der Problemlösung protokolliert wird, wozu z.B. auch eine Erfassung der Zeiten für einzelne Tätigkeiten gehört.

Die Ausarbeitungen sind bis jeweils spätestens zum untengenannten Termin abzugeben bzw. die Lösungen sind bis spätestens zu diesem Termin zu präsentieren.

8. Mai
29. Mai
19. Juni
10. Juli

Außerdem sei nochmal erwähnt, daß die Beteiligung bzw. das Stellen von Fragen bei den Besprechungen ausdrücklich *erwünscht* ist; davon hat nämlich sowohl der Frager, der Tutor als auch die Kommilitonen was.

2 Programmiersprachen

Im Praktikum sollen zwei Aufgaben mit Haskell, eine mit Java und eine mit Haskell und HTML programmiert werden. Haskell wird im Folgenden kurz vorgestellt, Java dann in Abschnitt 5.

2.1 Haskell

Haskell ist eine der zur Zeit wohl bedeutendsten, nicht-strikten, funktionalen Programmiersprachen. Wer sich eingehender in Haskell vertiefen möchte sei auf den Haskell-Report [PHA+97] verwiesen, in dem die Sprache definiert wird.
Zum Haskell-Programmieren stehen uns ein Interpretierer (`Hugs`) und verschiedene Übersetzer (`ghc`, `hbc`, `nhc`) zur Verfügung.

Hugs

Zunächst sei der `Hugs` erwähnt, der ja schon in der Vorlesung Praktische Informatik 1 verwendet worden ist. `Hugs` steht für Haskell Users Gofer System und es gibt ihn für verschiedene Rechner und Betriebssysteme: darunter Macs, DOS/Windows PCs oder Unix Rechner, wie die HPs hier an der Uni und die Linux PCs, die sich nicht zuletzt bei Informatik-Studenten wachsender Verbreitung erfreuen. Auf den HPs ist `Hugs` so installiert, daß mit dem Kommando `hugs` am Shell-Prompt der Interpreter gestartet wird.

hbc

Der Haskell B Compiler, hält sich an den Haskell-Report, erweitert diesen aber auf verschiedene (oft syntaktische) Arten. Der hbc ist aus dem LML Compiler entwickelt worden und ist selbst zum größten Teil in LML geschrieben. Er ist auf vielen Platformen, ebenso wie LML, verfügbar. An Prozessoren werden unterstützt: NS32k, M68K, I386, Alpha, ARM, Cray-1, HPPA, MIPS, PowerPC, RT/PC, SPARC, und Vax, an Betriebssystemen: OSF, HP-UX, FreeBSD, NetBSD, Linux, Irix, Aix, Solaris, SunOS... Der hbc kann auf den Rechnern des RBI mit dem Kommando `hbc` gestartet werden. Die Dokumentation des hbc steht als HTML Dateien unter `www.ki.informatik.uni-frankfurt.de/doc/hbc/` zur Verfügung.

ghc

Der Glasgow Haskell Compiler hält sich ebenso wie der hbc 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, anders als LML oder hbc, 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 und wir haben ihn auch schon in frühen Versionen auf den HPs des KI-Netzes zum Laufen gebracht. Heutzutage sind die HPs von ghc gut unterstützt. Für einige Plattformen gibt es zusätzlich einen Code-Erzeuger, dessen Verwendung die Übersetzungszeit reduziert, aber nicht zu so gut optimiertem Code führt wie die Verwendung des GNU C Übersetzers.

2.2 andere funktionale Sprachen

Gofer

Der Gofer Interpretierer von Mark P. Jones ist wohl als einer der Meilensteine für die Verbreitung von verzögert auswertenden funktionalen Sprachen anzusehen. Gofer ist schon seit den frühen 90ern verfügbar, leicht zu übersetzen, bietet als Interpretierer sagenhafte Übersetzungszeiten und war bevor es `Hugs` gab *das* Werkzeug für die Programmentwicklung in Haskell. Einer der Nachteile beim Benutzen von Gofer ist, daß Gofer in manchen Eigenschaften über Haskell hinausgeht und andere Eigenschaften von Haskell fehlen. Dieser Umstand hat dann zur Entwicklung von `Hugs` geführt.

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.

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.

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. Für sehr motivierte Studenten läßt sich eventuell ein gemeinsamer account auf den Rechnern des KI-Netzes einrichten, so daß man die Möglichkeit hat, auch Miranda auszuprobieren. Seit einiger Zeit gibt es auch eine Linux-Version von Miranda. Sie kostet allerdings eine Stange Geld.

[1] Miranda ist ein Warenzeichen von Research Software Limited

Clean

Clean [NSvP91] ist eine nicht-strikte funktionale Sprache, die an der Universität in Nijmegen entwickelt wurde. Sie ist für Suns, für Macs, für Linux, für Windows'95/NT und für OS/2 implementiert. Ursprünglich war Clean als Zwischensprache für einen Übersetzer aus einer höheren funktionalen Sprache (Miranda) entwickelt worden. Mittlerweile ist daraus aber selbst eine höhere funktionale Sprache geworden, die allen Komfort wie etwa Haskell auch bietet. Auch die Syntax ist ähnlich der von Haskell. Besonders ist an ihr das I/O Konzept, das es erlaubt, alle möglichen Fenster-, Maus- etc. Anwendungen zu programmieren und die Tatsache, daß der Clean Übersetzer wohl immer noch die kürzesten Übersetzungszeiten unter allen Übersetzern für verzögert auswertende funktionale Sprachen bietet. Eine starke Striktheitsanalyse [Nöc93] und die Ausnutzung der Ergebnisse bei der Übersetzung sorgt dafür, daß effizienter Code erzeugt wird. Eine ganze Reihe beeindruckender Beispielprogramme sind dem Programmpaket beigelegt.

2.3 Einführungsliteratur

Die meiste (ältere) Literatur über funktionales Programmieren bezieht sich auf Miranda. Man kann sich aber relativ einfach von der Syntax bei Miranda auf Haskell 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 theoretischer Hintergrund. Ein neueres Buch ist [Tho96], welches eine leichte Einführung darstellt, aber nicht sehr tiefgehend ist und keine fortgeschritteneren Programmiertechniken behandelt. Für anglophobe gibt es auch eine Einführung auf deutsch in [Hin92] sowie [Thi94]. Ein etwas älteres Buch noch aus Lisp-Zeiten ist [Hen80].

3 Benutzung von Hugs

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 und zwischen Code und Nicht-Code je eine Leerzeile steht. Die zweite Art, sogenannte *literate sources*, eignet sich, wenn man zu einem Programm viel Erklärungen hat. Dann kann das Programm im LaTeX-Format geschrieben werden, in dem der Code in der Verbatim-Umgebung steht. Der `Hugs` erkennt an der Endung des Dateinamens in welchem Modus die Datei verarbeitet werden soll: `.hs` Dateien werden als 'normaler' Quelltext betrachtet und `.lhs` Dateiten als *literate*. Dateien, die keine der beiden Endungen aufweisen, werden dann je nach dem, ob man `Hugs` mit `:set +l` in den *literate* Modus geschaltet hat oder nicht, als *literate* Quellen bzw. als 'normale' Quellen betrachtet.

3.2 Worte der Warnung

Haskell erspart dem Benutzer, Blöcke durch entprechende Worte oder Klammern zu markieren und aufeinanderfolgende Blöcke durch ein entsprechendes Zeichen zu trennen. Das hat aber seinen Preis: Die Aufgabe der Strukturierung übernehmen jetzt Einrückungsregeln. So kann z.B. die Blockstruktur eines Programmes durch Änderung der Einrückung verändert werden. Verletzt man diese Einrückungsregeln kann es zu Fehlermeldungen kommen, die ohne Kenntnis dieses Vorgangs 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 steht Kommentar.

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

Das hier ist Kommentar, der natürlich viel länger sein könnte...

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

Das Programm enthält Fehler. `Hugs` meldet beim Einlesen:

```
Reading file "usegofer.tex":
Parsing
ERROR "usegofer.tex" (line 215): Syntax error in input (unexpected `=')
```

Die 215. 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 werden aber die Semikolonzeichen und die Klammerung für die Blockstruktur eingefügt. 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 ver(sch)wenden 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-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`, `vi` und `emacs` können das.

- 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[2] zahlt sich dies langfristig aus.

[2] `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.

Im Hugs 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 Dies 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.

4 Haskell-Kurzübersicht

4.1 Funktionen und Operatoren

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 reservierte Wörter, dürfen nicht als Funktionsnamen verwendet werden:

```
case    class     data    default  deriving  do
else    if        import  in       infix     infixl
infixr  instance  let     module   newtype   of
then    type      where
```

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:

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

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, x ‘elem‘ xs`

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

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`.

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):

Vorrang	links-assoziativ	nicht-assoziativ	rechts-assoziativ
9	!!		.
8			^, ^^, **
7	*, /, ‘div‘, ‘mod‘, ‘rem‘, ‘quot‘		
6	+, -		
5		\\	:, ++
4		==, /=, <, <=, >, >=, ‘elem‘, ‘notElem‘	
3			&&
2			||
1	>>, >>=		
0			$, ‘seq‘

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

Typ–Namen

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

Der Typ-Operator (::)

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

42 :: Int 42 ist ein Integer-Wert

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

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)`

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.

Zeichen

Zeichenkonstanten werden durch Einschließen in Apostrophe (normale Quotes) dargestellt:

```
'a' :: Char
```

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 = "Hakell"
```

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)

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 Argument ist Funktion		zweites Argument ist Liste		Ergebnistyp ist Liste

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 Typkonstruktor `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 Typkonstruktor `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 Typkonstruktor mit einstelliger Konstruktorfunktion

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

einstelliger Typkonstruktor mit Konstruktorfunktion und Konstruktoroperator

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

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` repräsentiert 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!`

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 Kopf `x` und Schwanz `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`)

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:

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

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))

Unendliche Datenstrukturen

Verzögerte Auswertung erlaubt es in Haskell unendliche Datenstrukturen darzustellen. Z. B. könnte die unendliche Liste der Zweierpotenzen aller natürlichen Zahlen wie folgt dargestellt werden:

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

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

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

List Comprehensions

List comprehensions sind syntaktisch gut lesbare Darstellungen zur Listen-Verarbeitung. Die Grundstruktur von list comprehensions ist:

```
[ expr | qualifiers ]
```

wobei `qualifiers` eine der folgenden zwei 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 []`

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]
```

Lambda Ausdrücke

Lambda Ausdrücke gestatten die Definition namenloser Funktionen:

`\` *atomicPatterns* `->` *expr*

Beispiele für Lambda Ausdrücke:

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

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*

Irrefutable (verzögerte) Patterns

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:

> `(\(x, y) -> 0) undefined` = {undefined}
> Fehlermeldung, da `undefined` nicht die Form (x, y) hat. ('undefined' ist ein Haskellausdruck, 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.

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 Haskell-Programmen

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 beinhaltet ein Haskell Programm die Struktur durch Klammerung mit { und } (bei mehreren Blöcken, getrennt durch `;`):

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

Die Eingabe dieser Klammerung ist jedoch lästig und glücklicherweise 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 läßt 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). Dadurch, daß Tabs 8 Leerzeichen beim Einrücken entsprechen kann ein Problem werden, wenn z.B. der Editor so eingestellt ist, daß er für ein Tab nur 4 Zeichen einrückt.

4.6 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 `Char` (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 geschieht 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
        (x:xs) == (y:ys) = x == y && xs == ys
        _ == _           = False
```

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)

5 Java-Kurzübersicht

`Java` hat sich sehr schnell sehr weit verbreitet und es gibt eine Fülle an Büchern über `Java`. Diese Einführung ist daher nicht dazu gedacht die Sprache zu erlernen, sondern einen Überblick zu geben und auf geeignete Literatur zu verweisen.

Für alle jene `Java` schon kennen wird in dieser Übersicht nichts Neues mehr stehen, sie soll nur dazu dienen das Eis zu brechen und die Hemmschwelle sich mit einem der Bücher oder einem anderen Text zu `Java` auseinanderzusetzen.

5.1 Was ist `Java`?

`Java` ist eine Programmiersprache, die Objekt-Orientierung unterstützt. Programmieren in `Java` bedeutet daher das Definieren von Klassen und Methoden, sowie das Instantiieren der Klassen zu Objekten und das verwenden von Methoden. Die Klassen haben dabei eine ähnliche Bedeutung wie Typen in manchen prozeduralen Sprachen oder wie Typklassen in Haskell. Als weitere Einführung in `Java` sei an dieser Stelle die Broschüre zu `Java` vom RRZN der Uni Hannover empfohlen, die es für DM 6 beim HRZ zu kaufen gibt.

`Java` übersetzt den Programmtext in einen Zwischencode, wie das auch andere Übersetzer zunächst tun, welcher dann aber erst auf dem Zielsystem zur Ausführungszeit entweder vom Java-Byte-Code-Interpreter interpretiert wird oder aber von einem Just-In-Time-Compiler in tatsächlichen Maschinencode übersetzt wird. Die Verwendung des Java-Byte-Code-Interpreters ist davon die üblichere Vorgehensweise. Zwar entsteht durch das Interpretieren ein Geschwindigkeitsnachteil, aber wenn man bedenkt für welche Arten von Anwendungen `Java` entwickelt wurde, nämlich solche mit hohem Anteil an Netzwerk-Kommunikation und graphischer Benutzerinteraktion, so relativiert sich dieser Nachteil wieder.

Die Syntax von `Java` ist der von C++ sehr ähnlich, auch wenn bei der Entwicklung von `Java` darauf geachtet wurde eine ganz neue Sprache zu erhalten und solche Eigenschaften zu vermeiden, die immer wieder Probleme beim Programmieren verursacht haben. So gibt es, um nur ein Beispiel zu nennen, keine Operationen zum expliziten Referenzieren oder Dereferenzieren von Objekten oder Daten: primitive Daten werden immer dereferenziert verwendet und Objekte und Felder immer referenziert.

Das Fehlen von Referenzen und Zeigern, die vom Programm beliebig manipuliert werden können, sowie die Einschränkung der Zugriffsrechte für Applets, die über das Netzwerk kommen, tragen zur Sicherheit von `Java` bei.

5.2 Einige Beispiele

Um ein (erstes?) `Java`-Programm zu sehen und zu verstehen, betrachten wir folgendes Beispiel:

```
class Ausgabe
{ public static void main (String argv[])
  { System.out.println("Hallo Du da!");
} }
```

Dieses einfache Programm soll den Text "Hallo Du da!" ausgeben. Es besteht aus einer Klassendefinition, für die Klasse `Ausgabe`, die eine öffentliche Methode, `main`, zur Verfügung stellt. Das was innerhalb der {} der Klassendefinition steht sind die *Komponenten* der Klasse, also z.B. Komponenten zum Speichern von primitiven Daten, Methoden oder Konstruktoren. Die wichtigsten Zugriffsrechte für Komponenten einer Klasse sind `public` und `private`, wobei auf `public` Komponenten aus allen Klassen zugegriffen werden kann und auf `private` Komponenten nur von innerhalb der Klasse. Außerdem haben Datenkomponenten und Methoden einen expliziten (Rückgabe-)Typ, Konstruktoren haben dagegen als Rückgabe-Typ ein Objekt der Klasse. Der Rückgabe-Typ von `main` ist im Beispiel `void`, was besagt, daß die Methode `main` keine Werte zurückgibt. Die Methode `main` erwartet als Argument ein eindimensionales Feld von Objekten der Klasse `String`. Dieses Feld wird zur Laufzeit die Argumente enthalten, die beim Aufruf des Programms nach dem Programmnamen angegeben wurden. Im Körper der Methode `main` wird schliesslich die Methode `println` des Objektes `out` verwendet, welches seinerseits ein statisches Objekt (das entspricht etwa den globalen Variablen in prozeduralen Sprachen) der Klasse `System` ist.

Übersetzen und Ausführen

Beim Übersetzen des obigen Beispiels mit `javac Ausgabe.java` wird eine Datei `Ausgabe.class` erzeugt, die dann mit dem Kommando `java Ausgabe.class` ausgeführt werden kann und es erscheint als Ausgabe:

```
Hallo Du da!
```

Datenkomponenten

Datenkomponenten (entsprechen Variablen in prozeduralen Sprachen) müssen nicht wie bei C oder PASCAL am Anfang eines Blocks deklariert werden, es reicht, sie bei der ersten Zuweisung zu deklarieren.

Beispiel 1. `class Variable`
```
public static void main (String argv[])
{ String name = "Du da!";
  System.out.println ("Hallo "+name);
  }
}
```

In der dritten Zeile wird der Datenkomponente `name` der String `"Du da!"` zugeordnet und erklärt, daß `name` eine Referenz auf ein Objekt der Klasse `String` ist.

primitive Datentypen

Die primitiven Datentypen von `Java` sind:

`boolean`	entweder `true` oder `false`
`byte`	Zahlen von -128 bis 127 (8 Bit)
`char`	16-Bit Unicode-Zeichen
`short`	Zahlen von -32768 bis 32767 (16 Bit)
`int`	Zahlen von -2^{31} bis $2^{31}-1$ (32 Bit)
`long`	Zahlen von -2^{63} bis $2^{63}-1$ (64 Bit)
`float`	Fließkommazahlen
`double`	Fließkommazahlen mit doppelter Genauigkeit

Modifikatoren

Datenkomponenten, Methoden und Klassen können durch sog. Modifikatoren besondere Attribute zugewiesen bekommen. Ein solcher Modifikator, `final`, bewirkt beispielsweise, angewandt auf eine Datenkomponente, daß deren Wert als konstant angesehen wird und der Übersetzer diesen anstelle des Namens der Komponente verwenden kann.

Felder

Felder sind Anneinanderreihungen von Datenkomponenten gleichen Typs. Z.B. wird durch `String[] sprachen={"Haskell","Java","Pascal","Basic"};` ein eindimensionales Feld von Zeichenketten erzeugt und `sprachen` wird eine Referenz auf dieses Feld zugewiesen.

Ein einfaches Beispiel für ein zweidimensionales Feld wäre:

```
String[][] matrix = {
  {"AA","AB","AC","AD"},
  {"BA","BB","BC","BD"},
  {"CA","CB","CC","CD"}
};
```

Kommentare

Es gibt drei Arten von Kommentaren:

- Kommentar bis Zeilenende `//`
- Kommentare zwischen `/*` und `*/`
- kommentare zwischen `/**` und `*/`

Kommentare der letzten Art werden vom Programm `javadoc` aus dem Programmtext extrahiert, um zur Dokumentation der Klassen zu dienen.

Operatoren

Es gibt eine Reihe Operatoren, wie sie auch in anderen Sprachen zu finden sind, darunter arithmetische Operatoren, Vorzeichen, Vergleichsoperatoren, Zuweisungsoperator, logische Operatoren, Operatoren auf Bitmustern, Verschiebeoperatoren, Typwandlungsoperator und der Operator `instanceOf`. Mit dem Typwandlungsoperator kann man z.B. ein Objekt einer Unterklasse zum Aufruf einer Methode einer Oberklasse in ein Objekt dieser Oberklasse verwandeln, wenn ansonsten in der Unterklasse die Methode überdeckt wäre. Der `instanceOf` Operator ermöglicht es schliesslich festzustellen, ob ein Objekt zu einer bestimmten Klasse gehört.

Kontrollstrukturen

Die Kontrollstrukturen prozeduraler oder anderer objekt-orientierter Sprachen findet man auch in `Java`. Man verwendet geschweifte Klammern, um Blöcke zu bilden und diese dann als (zusammengesetzte) Anweisungen weiter zu verwenden. Die Kontrollstrukturen **`if`**, **`switch`**, **`while`**, **`do--while`**, **`for`**, **`break`**, **`continue`** und **`return`** stehen zur Verfügung. Bei den **`break`** und **`continue`** Anweisungen kann eine Marke angegeben werden, die den Block bezeichnet, der Verlassen werden soll bzw. mit dessen nächstem Duchlauf fortgefahren werden soll. Wird keine solche Marke angegeben, so beziehen sich **`break`** und **`continue`** immer auf den innersten Block bzw. die innerste Schleife.

Beispiel 2. `class break-continue`

```
{ public static void main (String args[])
  { byte a=0;
    byte b=0;
    while (a!=10)
    { System.out.println(a+" "+b);
      if (a==7)
        break;
      a++;
      if (b==5)
        continue;
      b++;
    }
  }
}
```


Die Ausgabe sieht so aus:

```
1 1
2 2
3 3
4 4
5 5
6 5
7 5
```

Nachdem die Datenkomponente `b` den Wert 5 erreicht hat, wird immer mit dem nächsten Schleifenduchlauf fortgefahren und `b` nicht weiter inkrementiert. Nachdem `a` den Wert 7 erreicht hat, wird die Schleife abgebrochen, obwohl die Bedingung noch erfüllt ist.

6 Parser und Parsergeneratoren

In der ersten Haskell-Aufgabe (Beweisen mit Hilfe von Wahrheitstabellen) soll der Parsergenerator `Lucky` für das Parsen des Eingabestrings verwendet werden. Dieses Kapitel beschreibt was ein Parser ist und motiviert die Verwendung von Parsergeneratoren in der Softwareentwicklung. Am Schluß wird auf die Benutzung des `Lucky` eingegangen.

6.1 Parser und Syntaxanalyse

Was ist ein Parser?

Wer einen Compiler für eine Programmiersprache entwickeln möchte, steht u.a. vor der Aufgabe eine Funktion zu schreiben, welche den Quellcode auf syntaktische Korrektheit überprüft. Der Quellcode ist zunächst nichts anderes als eine sehr lange Folge von ASCII-Zeichen ohne inhaltliche Bedeutung. Damit dieser String als ein korrektes Programm erkannt werden kann, muß die Anordnung der Zeichen gewissen Regeln genügen. Die Gesamtheit dieser Regeln wird als *Syntax* der Programmiersprache bezeichnet.[3] Ein Programm, welches die Syntaxanalyse durchführt, nennt sich *Parser*.

Man kann sich nun viele Beispiele ausdenken, wo auch in anderen Gebieten — also nicht nur im Bereich des Compilerbaus — Parser eingesetzt werden können. Ein solches Beispiel ist eben gerade die Überprüfung eines aussagenlogischen Ausdrucks. Ein anderes Beispiel möchte ich im nächsten Unterabschnitt vorstellen.

[3] Im Falle von `Haskell` kann man die Syntax im *Haskell Report* [PHA+97] nachsehen.

Beispiel: einfacher Taschenrechner

Wir wollen nun an einem einfachen Beispiel sehen, wie man zu einem gegebenen Problem eine exakte Syntaxbeschreibung angibt. Die Notation, welche wir hier verwenden, nennt sich *Backus Naur Form* oder kurz: *BNF*.

Die Syntax wird in Form von Ableitungsregeln angegeben. Dabei stehen auf der linken Seite einer Regel Variablennamen (*Nonterminals*), auf der rechten Seite stehen Strings aus weiteren Variablen und Symbolen aus dem Eingabestrom des Parsers (*Terminals*). Der Parser versucht nun ausgehend von einer Startvariablen solange Ableitungen durchzufuehren, bis ein String aus Terminals entstanden ist, der mit dem Eingabestring identisch ist (ähnlich einer Reduktionsfolge im λ-Kalkül).

Nehmen wir an, unsere Syntax (in BNF) lautete:

S	::=	a — a X
X	::=	X b — ϵ

wobei S das Startsymbol bezeichne, wir davon ausgehen, daß Nonterminals groß und Terminals klein geschrieben werden, das Zeichen «—» eine Art „Oder“ und ϵ das *„leere Terminal“* darstellt. Dies ist die Syntax der *formalen Sprache* $\{a, ab, abb, ...\}$. Die Ableitung des Wortes *abb* sieht dann z.B. wie folgt aus:

$$S \to aX \to a(bX) \to a(b(bX)) \to a(b(b\epsilon))$$

Als ein etwas komplexeres Beispiel betrachten wir einen Taschenrechner, welcher die vier Grundrechenarten beherrscht. Unser erster Ansatz wird wohl wie folgt lauten:

Expr	::=	Zahl
	—	[Expr]
	—	Expr + Expr
	—	Expr − Expr
	—	Expr ∗ Expr
	—	Expr / Expr
Zahl	::=	Ziff — Zahl Ziff
Ziff	::=	1 — 2 — ... — 9 — 0

d.h. Zahlen werden aus einzelnen Ziffern gebildet (wir verzichten auf negative Zahlen) und ein Rechenausdruck ist eine Zahl oder ist aus Unterausdrücken der gleichen Form zusammengesetzt. Dies sieht recht einleuchtend aus, aber es ist nicht wirklich das was wir wollen:

$$2 + 5 * 3$$

Dieser Eingabestring läßt sich verschiedentlich ableiten:

$$Expr \to Expr + Expr \to ... \to 2 + Expr \to 2 + (Expr * Expr) \to ...$$

$$Expr \to Expr * Expr \to ... \to Expr * 3 \to (Expr + Expr) * 3 \to ...$$

Die zweite Ableitung entspricht nicht dem, was wir von der Mathematik her kennen („Punkt-vor-Strich Regel“). Wir werden also diese zusätzlichen Regeln in irgendeiner Weise in unsere BNF einbauen müssen. Die folgende Syntax erfüllt die bekannten Präzedenzregeln:

Expr	::=	Expr + Term
	—	Expr − Term
	—	Term
Term	::=	Term ∗ Fact
	—	Term / Fact
	—	Fact
Fact	::=	[Expr]
	—	Zahl
Zahl	::=	Ziff — Zahl Ziff
Ziff	::=	1 — 2 — ... — 9 — 0

Es braucht ein wenig Übung zu verstehen, wie und warum dies funktioniert.[4]

6.2 Parsergeneratoren

Was ist ein Parsergenerator?

Nachdem man für eine Sprache eine Syntax in Form einer BNF ergestellt hat, geht es an die Umsetzung der Syntax in die Zielsprache. Je nach Komplexität der Syntax wird man diese Aufgabe irgendwo zwischen ‚lästig‘ und ‚unzumutbar‘ einordnen. Auch sieht man dem geschriebenen Quelltext oft kaum mehr die ursprüngliche Syntax an. An dieser Stelle kommen *Parsergeneratoren* ins Spiel.

Ein Parsergenerator ist ein Programm, welches als Eingabe eine Syntax in BNF[5] erhält und daraus einen Parser in der Zielsprache — in unserem Fall also Haskell — erzeugt. Dadurch konzentriert sich die Arbeit im wesentlichen auf das Finden einer Syntax, und die Fehler die auftreten sind i.a. logischer Natur (keine Programmierfehler).

Der bekannteste Parsergenerator ist `Yacc`,[6] welcher in den 70er Jahren für die Herstellung der Parser in UN*X C-Compilern entwickelt wurde und vielen anderen Parsergeneratoren (z.B. dem `Lucky`) zumindest was die Notation betrifft als Vorbild gedient hat. Eine ausführliche Einführung in die Benutzung dieses Software-Werkzeugs gibt [Her92]. In diesem Buch wird auch über das Programm `Lex` gesprochen. Dabei handelt es sich um einen Generator für sogenannte *Lexer* oder *Scanner*, welche sich um die *Lexikalische Analyse* kümmern, womit wir uns im nächsten Unterabschnitt beschäftigen werden.

[4] Da diese Herangehensweise auch für logische Ausdrücke (und damit für die erste Praktikumsaufgabe) gilt, ist es wichtig, daß man sich damit vertraut macht.
[5] bzw. einer der BNF ähnlichen Notation, wie im Beispiel des `Lucky`
[6] steht für **Y**et **a**nother **C**ompiler **C**ompiler :-)

lexikalische Analyse

Wir hatten gesagt, daß es sich bei der Eingabe eines Parsers um einen String von Zeichen handelt — üblicherweise sind dies ASCII-Zeichen. Dabei gibt es aber ein paar Dinge zu beachten. Betrachten wir dazu folgenden Haskell Text:

```
> func var = 5 * var
```

Es ist offenbar unerheblich, ob beispielsweise zwischen den Zeichen `5` und `*` kein, ein oder beliebig viele Freizeichen stehen. Auf der anderen Seite ergibt sich ein völlig anderes Programm, wenn wir schreiben:

```
> func v ar = 5 * v ar
```

Wir müssen also unterscheiden zwischen den ASCII-Zeichen, wie sie uns im Eingabestring begegnen und den Zeichen, die aus der Sicht der Syntax *Sinneinheiten* bilden. Solche Sinneinheiten sind im obigen Beispiel die „Zeichen“ `func`, `var`, `=`, `5`, `*` und `var`. *Keine* Sinneinheiten sind dagegen die Zwischenräume und der abschließende Wagenrücklauf (`'\n'`).[7] Man nennt diese Sinneinheiten *Token* — es ist die Aufgabe eines Lexers, aus dem Strom von ASCII-Zeichen eine Folge von Token zu generieren, um den Parser von technischen Details zu entlasten. Eine solche Tokenfolge könnte für obiges Beispiel wie folgt aussehen:[8]

```
[BEZEICHNER func, BEZEICHNER var, ZUWEISUNG, INT 5,
MULT, BEZEICHNER var]
```

Eine lexikalische Analyse ist nun ein *Pattern Matching* ähnlich dem, wie wir es von Haskell her kennen. D.h. auf der linken Seite stehen Muster, denen gewisse Folgen von ASCII-Zeichen entsprechen können. Diese Muster heißen reguläre Ausdrücke — mit ihnen beschäftigt man sich auch in der Praktischen Informatik 2 und ausgiebig in der Theoretischen Informatik 2.[9] Auf der rechten Seite stehen Token, die im Falle eines Matchings erzeugt werden sollen. Einige Beispiele sollen dies klarmachen:

$(\texttt{' '} + \texttt{'\textbackslash t'} + \texttt{'\textbackslash n'})^+$	ϵ
`*`	`MULT`
$(\text{a-z})^+$	`Bezeichner $$`

Die erste Zeile in diesem Beispiel ordnet einer Folge von einem oder mehreren[10] Whitespace-Charactern das „leere Token“ — üblicherweise durch ϵ dargestellt — zu.[11] Dies bedeutet effektiv, daß diese Zeichen schlicht überlesen werden sollen. Die zweite Zeile ordnet dem Zeichen ‚`*`‘ das entsprechende Token `MULT` zu. In der dritten Zeile werden

[7] Diese Zeichen werden auch als „Whitespace“ bezeichnet.

[8] wobei wir gleich eine Schreibweise wählen, wie wir sie auch in Haskell benutzen würden, vorausgesetzt ein Datentyp `Token` existierte.

[9] Wer sich mit UN*X auskennt, dem dürften reguläre Ausdrücke von Programmen wie `grep` oder den Suchfunktionen verschiedener Texteditoren bekannt sein.

[10] dies ist die Bedeutung des ‚$^+$‘ am Ende der linken Seite

[11] Ein ‚+‘ entspricht einem logischen Oder

Bezeichner aus Kleinbuchstaben erkannt. `$$` steht hierbei für den Teilstring, der gematcht wurde, also z.B. `func` oder `var`.

Das Programm `Lex` ist wie bereits erwähnt ein Lexer-Generator. Da man aber den ASCII-Strom häufig recht einfach „von Hand“ in den zugehörigen Tokenstrom überführen kann, wird ein solches Programm seltener benutzt als ein Parsergenerator.

6.3 `Lucky`

`Lucky` von Norbert Klose ist ein Parsergenerator, dessen Zielsprache Haskell ist. Er orientiert sich in der Notation am `Happy`, ein weiterer Haskell-Parsergenerator, welcher seinerseits viel von `Yacc` übernommen hat. Dies bezieht sich insbesondere auf die interne Funktionsweise des `Happy` — für den `Lucky` wurde ein anderer Aufbau gewählt, worauf aber hier nicht eingegangen werden soll.[12]

Die offizielle Referenz ist [Klo97]. Sie kann entweder von Norberts Homepage[13] bzw. von der Seite des Praktikums bezogen werden. Dieses Manual ist leider etwas knapp ausgefallen, weil sich `Lucky` in den meisten syntaktischen Fragen eng an `Happy` hält und demzufolge auf dessen Anleitung verweist. Das auf Seite 10 angegebene Beispiel für ein Lucky-Skript ist allerdings recht nützlich.[14]

Einen Lexer-Generator gibt es zwar auch für Haskell (`Alex`), aber es zeigt sich, daß es gerade in Haskell durch dessen ausgefeiltes Patternmatching selten nötig ist, eine solche Software zu verwenden. Stattdessen schreibt man einen Lexer gewöhnlich von Hand als ein eigenständiges Modul und bindet dieses mit dem Befehl `\import` in das `Lucky`-Skript ein (s. nächsten Unterabschnitt).

Aufbau eines `Lucky`-Skripts

Jedes `Lucky`-Skript besteht aus bis zu vier Teilen. Der erste (optionale) Teil ist ein Block Haskell-Code, der von geschweiften Klammern umschlossen wird. Dieser Block wird unverändert an den Anfang der durch `Lucky` generierten Datei gesetzt. Für gewöhnlich stehen hier Typdeklarationen, `\import`-Befehle usw. Der nächste Teil enthält verschiedene Direktiven, die `Lucky` für eine korrekte Funktionsweise unbedingt benötigt:

- ≪`%name` *NAME*≫ bezeichnet den Namen der Parserfunktion. Unter diesem Namen kann der Parser also später aufgerufen werden.
- ≪`%tokentype` { *TYPE* }≫ Dies ist der Ausgabetyp des Lexers und damit der Eingabetyp des Parsers.

[12] Ich habe dies ohnehin nur erwähnt, damit man sich nicht wundert, warum es gleich zwei Parsergeneratoren für Haskell gibt.

[13] (http://www.ki.informatik.uni-frankfurt.de/ klose)

[14] ... und die deutsche Übersetzung der *GNU General Public License (GPL)*, unter die `Lucky` gestellt wurde, darf man sich auch gerne einmal durchlesen :)

- ≪`%token` *MATCHLIST*≫ Hier werden den Token, die vom Lexer erzeugt wurden, die *Terminals* zugewiesen, die in der BNF verwendet werden.[15]

Es schließt sich der Grammatikteil an (vom zweiten Teil durch ein `%%` getrennt), in dem also in einer BNF ähnlichen Notation die Syntax, wie man sie sich zuvor überlegt hat, aufgeschrieben wird. Auf die kleinen Unterschiede zur BNF möchte ich hier nicht eingehen, wichtiger ist, daß man hinter jede Regel eine sogenannte Aktion schreiben kann. Als Beispiel diene:

```
> start    : {- empty -}   { [] }
>          | start ATOM    { show $2 : $1 }
```

Zunächst bezeichnet `{- empty -}` das leere Wort — dies ist aus Lesbarkeitsgründen Konvention. Hinter den Regeln steht in geschweiften Klammern jeweils ein Stück Haskell-Code. Dies sind „Aktionen“, die immer dann ausgeführt werden, wenn diese Regel abgeleitet wird. Mittels `$1` wird auf den Wert von *start* und mittels `$2` auf den Wert von ATOM zugegriffen. Der Wert eines Terminals ist dabei normalerweise das Terminal selbst. Durch die Aktionen hat der Parser also eine Ausgabe (und ist nicht nur ein reiner Syntax-Überprüfer). In unserem Beispiel ist die Ausgabe eine Liste von "ATOM“-Strings.

Der vierte Teil eines `Lucky`-Skripts ist wieder ein in geschweifte Klammern gesetzter Block mit Haskell-Code, welcher unverändert ans Ende der erzeugten Datei gesetzt wird. Hier muß zumindest die Funktion `luckyError` stehen, welche im Fall eines Syntax-Fehlers von der Parser-Funktion automatisch angesprungen wird (damit dies funktioniert, darf für diese Funktion kein anderer Name verwendet werden.) Oft ist hier auch der Lexer implementiert, wenn der Programmierer zu faul war, ihn in ein eingenes Modul zu stecken.

Ich möchte noch einmal auf das Beispiel in [Klo97] verweisen, welches sehr gut für das Verständnis des eben Besprochenen geeignet ist.[16]

7 Aufgabe 1: Aussagenlogischer Beweiser in Haskell

In der ersten Aufgabe sollen aussagenlogische Beweiser in Haskell geschrieben werden. Die Zielfunktionalität ist ein Programm, das eine textuelle Darstellung einer aussagenlogischen Formel einliest und überprüft, ob es sich dabei um eine Tautologie handelt. Als Ausgabe soll eine textuelle Darstellung des Beweises generiert werden. Es gliedert sich die Aufgabe dabei in folgende Teilaufgaben:

- ein Parser
- ein Beweiser über Wahrheitstafeln

[15] Warum man hier noch einmal zusätzlich zwischen Token und Terminals unterscheidet, verstehe ich nicht ganz. Der Grund könnte sein, daß auf diese Weise evtl. komplexe Token, die Zusatzinformationen wie zB. Positionsangaben beinhalten (s. Beispiel in [Klo97]), auf diese Weise abgekürzt werden können.

[16] Hier ein eigenes Beispiel zu bringen, würde den Rahmen dieses Skriptes sprengen.