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

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

Übung: funktionales Programmieren

Wintersemester 1995/96

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

6. November 1995

Inhaltsverzeichnis

1 Organisation

1.1 Sprechstunden

Während des Semesters wird Norbert Klose eine Sprechstunde haben, in der er bei Fragen oder Schwierigkeiten im Raum 026 zur Verfügung stehen wird.

Norbert Klose	?	? Uhr

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

Norbert Klose	klose@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 wir denken, 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. Die Übungsblätter werden von Norbert mitentworfen und korrigiert. Die Aufgaben werden sowohl theoretischer als auch praktischer Natur in Form kleiner Programme sein. Programme, die eine gewisse Länge überschreiten, sind Norbert während seiner Sprechzeit vorzuführen. Näheres wird bei den entsprechenden Aufgaben angegeben.

2 Programmiersprachen

Die Vorlesung wird die ganze Breite der funktionalen Programmiersprachen abdecken. Dieses beinhaltet damit sowohl strikte als auch lazy Sprachen, Sprachen mit und ohne Typsystem, reine funktionale Sprachen und Sprachen mit destruktiven (speicherverändernden) Befehlen.
Für die Übung stehen uns Scheme und Gofer zur Verfügung.

2.1 Scheme

Scheme ist eine strikte, dynamisch getypte funktionale Sprache höherer Ordnung mit Seiteneffekten. Scheme gibt es für verschiedene Rechner und Betriebssysteme: darunter Atari, Amiga, DOS PCs oder Unix Rechner. Auf den HPs ist Scheme so installiert, daß mit dem Kommando `scheme` am Shell-Prompt der Interpreter gestartet wird. Scheme ist ein Dialekt der verbreitesten funktionalen Sprache Lisp [Ste90]. Zu Scheme gibt es ein Einführungsbuch [AS85], welches in ausreichender Zahl in der Bibliothek auch auf Deutsch vorhanden ist. Es gibt für den Scheme-Interpreter, den wir benutzen, auch ein Sprachmanual, welches als Kopiervorlage (paperware) in der Bibliothek vorhanden ist.

2.2 Gofer

Gofer gibt es für verschiedene Rechner und Betriebssysteme: darunter Macs, DOS PCs oder Unix Rechner, wie die HPs hier an der Uni und die Linux PCs, die einige möglicherweise verwenden.
Auf den HPs ist Gofer so installiert, daß mit dem Kommando `gofer` am Shell-Prompt der Interpreter gestartet wird. Gofer implementiert einen großen Teil von Haskell, einer, zur Zeit wohl der bedeutendsten, nicht-strikten, funktionalen Programmiersprache. Wer sich eingehender in Haskell vertiefen möchte sei auf den Haskell-Report [HPW+92] verwiesen, in dem die Sprache definiert wird.
Für den Goferinterpreter gibt es auch ein Sprachmanual, welches als Kopiervorlage (paperware) in der Bibliothek vorhanden ist.

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 von uns 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 entsprechendem 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. 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. Es gibt auch eine Linux-Version von Miranda. Sie kostet allerdings eine Stange Geld.

LML LML steht für <u>L</u>azy <u>M</u>L und implementiert die Programmiersprache ML nicht-strikt im Gegensatz zu SML, was eine strikt auswertende Umsetzung von ML ist.

HBC Der <u>H</u>askell <u>B</u> <u>C</u>ompiler 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 entwickelt worden und ist auf vielen Platformen, ebenso wie LML, verfügbar. Leider sind beide nicht auf HP-PA RISC Architekturen verfügbar. Es gibt hingegen eine Linux-Version.

[1] Miranda ist ein Warenzeichen von Research Software Limited

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. Für sehr motivierte Studenten gilt wieder das unter Miranda Geschriebene. Bei entsprechendem Bedarf können wir den GHC auch auf das RBI-Netz installieren lassen, was allerdings mit einigem Aufwand verbunden ist.

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.

hugs Hugs ist im Kern derselbe Interpreter wie Gofer mit dem Unterschied, daß sich hugs an den Haskell-Report hält.

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 erreicht (was Sprachsyntax und Ausdrucksmächtigkeit betrifft). Haskellprogramme lassen sich somit schnell eins-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 man auch ein bißchen mehr theoretischen Hintergrund. Für anglophobe gibt es auch eine Einführung auf deutsch in

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

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

3 Programmieren in Scheme und Gofer

3.1 Arbeiten mit einem Interpreter

Beim Arbeiten mit einem Interpreter wie Scheme oder Gofer empfiehlt sich die folgende Vorgehensweise:

1. Starten eines Editors (z.B. `me`) innerhalb eines Terminalfensters (z.B. `XTerm`, `HPTerm`) oder starten eines anderen Editor Programmes. Hier wird der Source-Code geschrieben und bearbeitet.
2. Starten des Interpreters (z.B. `Scheme`) innerhalb eines weiteren Terminalfensters.
3. Laden der Textdateien in den Editor
4. Testen der Programme im Read-Eval-Print-Loop des Interpreters

3.2 Wahl des Editors

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

- Gofer 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` wissen wir 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 allem die Klammern richtig zählt. In der Schemeinstallation ist eine emacs-Anpassung für Scheme namens edwin enthalten. Hierzu ruft man `scheme -edwin` auf.
- 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.

[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 le-

- Die neueste Version von `emacs`, die 19.28, hat Ralf Zessin vor kurzem installiert.
- emacs ist übrigens in Lisp geschrieben. Also ein Beispiel für ein großes in einer funktionalen Sprache enwickeltes Programm.

Im Gofer-Interpretierer gibt es das `:edit`-Kommando, das direkt den für Gofer 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 Gofer zu verwenden.

3.3 Scheme

3.3.1 Starten und Beenden von Scheme

Scheme ist auf den HP's des RBI installiert und wird in einem Terminalfenster durch Eingabe von

```
>  scheme
```

gestartet. Danach befindet man sich im Read-Eval-Print-Loop (REPL) angezeigt durch ein Prompt

```
]=>
```

Jetzt kann man mit seiner Arbeit beginnen. Will man Scheme wieder verlassen, dann gibt man

```
]=> (exit)
```

ein.

3.3.2 Laden von Datein

Datei werden mit dem Befehl

```
]=> (load filename)
```

in den Interpreter geladen. Achten sollte man dabei darauf, daß man den Interpreter aus einem Verzeichnis heraus startet, aus dem die Dateien gefunden werden können. Source-Code Dateien, sollten die Standardmäßige Endung `".scm"` haben. Das aktuelle Arbeitsverzeichnis in dem sich Scheme befindet, kann mit

```
(pwd)
(cd pathname)
```

abgefragt bzw. gewechselt werden. Die Parameter `filename` bzw. `pathname` sind String Parameter und müssen daher in Gänsefüßchen eingeschlossen werden.

sen/schreiben, EMail lesen/schreiben, aus ihm heraus TeX oder einen Previewer starten, Lisp Programme schreiben und laufen lassen und vieles mehr.

3.3.3 Fehlermeldungen

Entdeckt der Interpreter einen Fehler, stopt er den momentanen REPL, gibt eine Fehlermeldung aus und startet einen neuen REPL.

```
1 ]=> foo

Unbound variable foo

2 Error->
```

Die 2 vor dem Prompt zeigt an, daß man sich in einem neuen REPL befindet. Um in den Top-Level-Loop zurückzukehren, gibt man `C-g` (C steht für CTRL, M für META bzw. ESC) ein.

3.3.4 Tastatur-Codes

`C-c`	Es wird ein weiters Zeichen erwartet. Die Eingabe von ? liefert eine Liste möglicher Aktionen.
`C-c C-x`	Stopt die momentane Auswertung und kehr zum aktuellen REPL zurück.
`C-c C-u`	Wie `C-c C-x`, nur das ein REPL nach oben gesprungen wird.
`C-g`	Stopt die momentane Auswertung und kehrt zurück zum Top-Level.

3.4 Gofer

3.4.1 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 Gofer-Interpretierer wird durch die Kommandos `:set -l` bzw. `:set +l` von dem einen in den andren Modus umgestellt.

3.4.2 Worte der Warnung

Gofer 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 Kommntar.

```
> 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. Gofer 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 Gofer, 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.

3.5 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.
Unserer 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.

4 Scheme-Kurzübersicht

Die folgende Scheme Kurzeinführung bezieht sich auf das *MIT Scheme Reference Manual, Edition 1.1 for Scheme Release 7.1.3, November 1991* [HT91]. Von dort stammen auch einige der Beispiele. Eine Ausgabe steht in der Bibliothek zum Kopieren zur Verfügung.

4.1 Lexikalische Vereinbarungen

4.1.1 Zeichen und Zeichenfolgen mit besonderer Bedeutung

Whitespace: Hierzu zählen `space`, `newline`, `tab` und `page break`. Sie dienen zum Trennen der einzelnen Tokens und zum Strukturieren von Programmcode.

Klammern: Mit Klammern `( )` werden Ausdrücke verschachtelt und Listen dargestellt.

Kommentare: Ein Semikolon `;` leitet einen einzeiligen Kommentar ein. Alles hinter einem Semikolon in einer Zeile wird ignoriert.

```
#|
Damit kann mehrzeiliger Kommentar eingeschlossen werden.
|#
```

Strings: Gänsefüßchen `"` begrenzen String-Konstanten (z.B. `"Hallo Welt."`).

Quoting: Single Quotes `'` unterdrücken die Auswertung von Objekten. Übergibt man einer Prozedur eine Liste `(a b)`, dann versucht der Interpreter die Ausdrücke `a` und `b` auszuwerten. Übergibt man stattdessen `'(a b)`, dann wird die Auswertung unterdrückt. Backquotes `` ` `` werden im Zusammenhang mit Quasiquote verwenden (siehe Quasiquote).

Characters: Character-Konstanten werden durch `#\` dargestellt (z.B. `#\a`).

Zwischen Groß- und Kleinschreibung wird in Scheme nicht unterschieden.

4.1.2 Identifier

Ein Identifier ist eine Zeichenfolge, die keines der obengenannten Zeichen enthalten darf. Da man ansonsten recht frei ist, was die Wahl von Bezeichnern angeht, sollte man durchaus Gebrauch davon machen und Bezeichner so wählen, daß sie eine möglichst starke Aussagekraft besitzen.

4.1.3 Reservierte Zeichenfolgen

Folgende Zeichenfolgen haben in Scheme eine besondere Bedeutung und sollten nicht als Identifier verwendet werden:

`access`	`and`	`begin`	`case`
`cond`	`cons-stream`	`declare`	`define`
`delay`	`do`	`error`	`if`
`lambda`	`let`	`let*`	`letrec`
`or`	`quasiquote`	`quote`	`sequence`
`set!`			

4.2 Spezialformen

Spzialformen sind solche Konstrukte, die einer besonderen Auswertungsreihenfolge unterliegen.

4.2.1 Lambda und Define

Der λ-Kalkül ist eine formale Sprache, um Funktionen notieren und auswerten zu können, die allen funktionalen Programmiersprachen zugrunde liegt. Eine Funktion $f : X_1 \times \ldots \times X_n \to Y$ notieren wir im λ-Kalkül als

$$\lambda x_1.(\cdots.\lambda x_n.e),$$

wobei dem Ausdruck e ein Wert aus Y entspricht. In Scheme können wir diesen Ausdruck fast direkt übersetzen. So liefert

```
(lambda (x1 x2) (+ x1 x2))
```

eine Funktion, die zwei Parameter erwartet. Wir können nun einen Funktionsaufruf auswerten lassen:

```
]=> ((lambda (x1 x2) (+ x1 x2)) 1 2)
3
```

Um obige Definition von Funktionen zu erleichtern, gibt es in Scheme die Spezialform `define`:

```
(define (f x1 x2) (+ x1 x2))
```

Der Vorteil hiervon ist, daß die so definierte Funktion an ein Funktionssymbol gebunden wird, das wir von nun an anstelle des `lambda` Konstruktes benutzen können:

```
]=> (f 1 2)
3
```

4.2.2 Let

Mit `let` können wir Symbolen lokale Werte zuweisen. Zusätzlich können kompliziertere Programme dadurch übersichtlicher gestaltet werden.

```
(let ((var1 init1)
      ...
      (varN initN))
     expr1 ... exprN)
```

Bei dieser Spezialform ist zu beachten, daß die Werte von `init1` bis `initN` erst an die Symbole `var1` bis `varN` gebunden werden, nachdem alle `init`'s ausgewertet wurden. So liefert z.B.:

```
]=> (define v1 3)
v1

]=> (define f
      (let ((v1 2)
            (v2 v1))
```

```
           (+ v1 v2)))
f

]=> f
5
```

Soll `v1` bereits zur Auswertung von `v2` an den neuen Wert gebunden sein, so müssen wir die Spezialform `let*` verwenden:

```
]=> (define v1 3)
v1

]=> (define f
      (let* ((v1 2)
             (v2 v1))
            (+ v1 v2)))
f

]=> f
4
```

Wir können auch `define` für interne Definitionen verwenden.

```
]=> (define v1 3)
v1

]=> (define (f)
      (define v1 2)
      (define v2 v1)
      (+ v1 v2))
f

]=> (f)
4
```

4.2.3 Quote und Quasiquote

Wie oben bereits erwähnt, kann die Auswertung von Ausdrücken durch Voranstellen eines Single Quotes `'` vor dem betreffenden Ausdruck unterdrückt werden. Das Gleiche leistet auch `quote`:

```
]=> (quote (list (+ 1 2) 4))
(list (+ 1 2) 4)

]=> '(list (+ 1 2) 4)
(list (+ 1 2) 4)
```

Wenn innerhalb des Ausdrucks `(list (+ 1 2) 4)` noch der Term `(+ 1 2)` ausgewertet werden, aber der Rest erhalten bleiben soll, sprechen wir von Quasiquote. Dies erreichen wir durch ein Voranstellen eines Backquotes ‘ vor den Ausdruck, der nicht ausgewerten werden soll. Wollen wir einen Subterm auswerten, so stellen wir diesem ein Komma voran:

```
]=> `(list ,(+ 1 2) 4)
(list 3 4)

]=> (quasiquote (list (unquote (+ 1 2)) 4))
(list 3 4)
```

4.2.4 Bedingte Auswertung

Eine Bedingte Auswertung erhalten wir durch `if` oder `cond`.

```
(if (predicate)
    consequent alternative)

(cond (predicate1 expression1 ...)
      ...
      (predicateN expressionN ...)
      (else expression ...))
```

Bei der Auswertung von `cond` werden zuerst der Reihe nach von oben nach unten die Prädikate ausgewertet, bis eine Auswertung schließlich `#t` liefert. Dann werden die entsprechenden `expression`'s von rechts nach links ausgewertet. Der Wert des letzten Ausdrucks wird zurückgeliefert. Liefern alle Prädikate `#f` und ist keine `else` Klausel vorhanden, dann ist der Wert von `cond` *unspezifiziert*.

4.2.5 Case

```
(case key
      ((object1 ...) expression1 ...)
      ...
      ((objectN ...) expressionN ...)
      (else expresiion ...))
```

Bei einer `case`-Anweisung wird zuerst der Schlüssel `key` ausgewertet. Dann werden der Reihe nach von oben nach unten die `object` Ausdrücke ausgewertet. Stimmt ein `object` mit dem `key` überein, dann werden die entsprechenden `expression`'s von rechts nach links ausgewertet. Der Wert des letzten Ausdrucks wird zurückgeliefert. Liefern alle Vergleiche `#f` und ist keine `else` Klausel vorhanden, dann ist der Wert von `case` *unspezifiziert*. Ein Vergleich ist `#t` im Sinne von `eqv?` (siehe `eqv?`).

```
]=> (case (* 2 3)
      ((2 4 6) "even")
      ((1 3 5) "odd"))
"even"
```

4.3 Gleichheiten

4.3.1 eqv?

```
(eqv? object1 object2)
```

`eqv?` liefert `#t`, falls eine der folgenden Bedingungen eintritt:

- `object1` und `object2` sind beide `#t`, `#f` oder `()`.
- `object1` und `object2` bezeichnen die gleichen Symbole.
- `object1` und `object2` sind gleich im Sinne von `=`.
- `object1` und `object2` bezeichnen die gleiche Prozedur (ohne lokale Variablen!)
- `object1` und `object2` haben den gleichen Typ und referenzieren die gleiche Speicherzelle

```
]=> (eqv? 'a 'a)
#t

]=> (eqv? '(a) '(a))
#f

]=> (let ((p (lambda (x) x))) (eqv? p p))
#t
```

4.3.2 eq?

```
(eq? object1 object2)
```

`eq?` liefert angewendet auf Symbole, Booleans, `()`, Paaren und Strings das gleiche Ergebnis wie `eqv?`. *Gedanklich* vergleicht `eq?` die Speicherreferenzen von `object1` und `object2` und kann daher mit konstanten Aufwand implementiert werden.

4.3.3 equal?

```
(equal? object1 object2)
```

`equal?` vergleicht rekursiv die Datenstruktur von `object1` und `object2`. *Gedanklich* liefert `equal? #t`, wenn beide Objekte gleich ausgedruckt werden. Aufpassen bei zyklisch verketteten Datenstrukturen!

```
]=> (equal? 'a 'a)
#t

]=> (equal? '(a) '(a))
#t
```

4.4 Elementare Datentypen

4.4.1 Numbers

Scheme kennt die numerischen Datentypen mit folgender Hierarchie

```
number
complex
real
rational
integer
```

und entsprechenden Prädikaten

```
(number? object)
(complex? object)
(real? object)
(rational? object)
(integer? object)
```

```
]=> (complex? 3+4i)
#t

]=> (real? 3)
#t

]= (integer? 3+0i)
#t
```

Folgende weitere Prädikate und Funktionen zur numerischen Auswertung stehen zur Verfügung

```
(= z1 z2 z3 ...)
(< z1 z2 z3 ...)
(> z1 z2 z3 ...)
(<= z1 z2 z3 ...)
(>= z1 z2 z3 ...)
(zero? z)
(positive? z)
(negative? z)
(odd? z)
(even? z)
(max z1 z2 z3 ...)
(min z1 z2 z3 ...)
(+ z1 z2 z3 ...)
(* z1 z2 z3 ...)
(- z1 z2 z3 ...)
(/ z1 z2 z3 ...)
(1+ z)
(-1+ z)
(number->string number)
(string->number string)
abs quotient remainder modulo exp log sin cos tan asin acos
atan sqrt expt real-part imag-part conjugate make-rectanglar
```

```
]=> (quotient 7 3)
2

]=> (remainder -13 4)
-1

]=> (modulo -13 4)
3
```

4.4.2 Characters

Wie oben beschrieben werden Characters durch das Präfix `#\` repräsentiert. Characters mit speziellen Codes sind z.B. folgende:

```
#\space
#\newline
#\linefeed
#\return
#\backspace
#\page
#\tab
#\c-a         ; Control-a
#\meta-b      ; Metra-b
...
```

Nützliche Prädikate und Funktionen sind z.B.

```
(char? object)
(char=? char1 char2)
(char<? char1 char2)
(char>? char1 char2)
(char<=? char1 char2)
(char>=? char1 char2)
(char-upcase char)
(char-downcase char)
(char->digit char)
(digit->char digit)
(char-code char)
(integer->char char)
(char-upper-case? char)
(char-lower-case? char)
(char-alphabetic? char)
(char-numeric? char)
(char-alphanumeric? char)
(char-whitespace? char)
(char-graphic? char)
(char-standard? char)
```

4.4.3 Strings

Strings werden, anders als in Gofer, als Bytefolge – eingeschlossen in Gänsefüßchen – dargestellt. Innerhalb eines Strings werden die folgenden Zeichencodes besonders behandelt:

```
\" \t \n \f \\
```

Nützliche Prädikate und Funktionen sind z.B.

```
(string? object)
(list->string char-list)
(string->list string)
(string-length string)
```

```
(string-null? string)
(string-ref string k)
(string=? string1 string2)
(string-downcase string)
(string-upcase string)
(string-append string1 string2)
```

4.4.4 Paare

Ein Paar ist ein abstrakter Datentyp mit zwei Feldern `car` und `cdr` (sprich "codder"), einem Konstruktor `cons` und zwei Selektoren `car` bzw. `cdr`. In Scheme werden Paare meist in der *dotted* Notation dargestellt:

```
]=> (cons 1 2)
(1 . 2)

]=> (car (1 . 2))
1

]=> (cdr (1 . 2))
2
```

Nützliche Prädikate und Funktionen sind z.B.

```
(pair? object)
caar cadr cddr cdar ...

]=> (cadr (1 . (2 . 3)))
2
```

4.4.5 Listen

Eine Liste ist eine rekursive Datenstruktur. So ist die leere Liste `()` eine Liste. Ist l eine Liste, dann ist l ein Paar, dessen `cdr`-Feld wieder eine Liste ist. Anders als in Gofer ist es erlaubt, Objekte verschiedenen Types in einer Liste unterzubringen. Es liegt dann beim Programmierer, die Liste mit entsprechenden Typchecks durchzuarbeiten. Um Listen besser darstellen zu können, sind folgende Notationen äquivalent:

```
]=> (list 1 2 3 4)
(1 2 3 4)

]=> (cons 1 (cons 2 (cons 3 (cons 4 ()))))
(1 . (2 . (3 . (4 . ()))))
```

Nützliche Prädikate und Funktionen sind z.B.

```
(list? object)
(length list)
(null? object)                ; #t, falls (eq? object '())
(list-ref list k)
(append list ...)
(reverse list)
```

```
]=> (append '(a b c) '(d e f))
(a b c d e f)
```

```
  (memq object list)          ; vergleicht mit eq?
  (memv object list)          ; vergleicht mit eqv?
  (member object list)        ; vergleicht mit equal?
```

```
]=> (memq 'b '(a b c))
(b c)

]=> (memq 'd '(a b c))
()
```

4.4.6 Booleans

```
  (boolean? object)
  (not bool)
  (and bool1 bool2)
  (or bool1 bool2)
```

Es ist in Scheme üblich, Prädikatensymbole mit einem abschließenden Fragezeichen zu versehen (z.B. equal?).

4.5 Funktionen höherer Ordnung

Funktionen höherer Ordnung, sind Funktionen, die andere Funktionen als Parameter aufnehmen, oder neue Funktionen dynamisch erzeugen (siehe `lambda`). Dazu ein kleines Programmbeispiel:

```
]=> (define id (lambda (x) x))
id

]=> (define (summe term von bis naechster)
       (if (> von bis)
           0
           (+ (term von)
              (summe term (naechster von) bis naechster))))
summe

]=> (summe id 1 100 1+)
5500

]=> (summe (lambda (x) x) 1 100 1+)
5500
```

Beispiele von – in Scheme vorhanden – Funktionen höherer Ordnung sind

```
  (map procedure list)
  (for-all? list predicate)
```

`map` wendet `procedure` auf jedes Listenelement von `list` an und liefert die Ergebnisliste zurück. `for-all?` liefert `#t`, falls `predicate` angewendet auf jedes Listenelement `#t` ergab. Liefert eine Auswertung von `predicate` angewendet auf ein Listenelement `#f`, wird die Auswertung abgebrochen und `#f` zurückgeliefert.

```
]=> (map 1+ '(1 2 3))
(2 3 4)

]=> (for-all? '(2 4 6) even?)
#t
```

4.6 Ausgabe Routinen

Zur Ausgabe von Programmnachrichten auf das Terminalfenster stehen folgende Prozeduren zur Verfügung:

```
(write object)
(display object)
(pp object [as-code?])
(newline)
```

Der Unterschied zwischen `write` und `display` ist der, daß `write` eine maschinen internere Repräsentation eines Objektes ausgibt, als `display` dies tut. So werden Strings mit `write` z.B. in Gänsefüßchen ausgegeben, während `display` den reinen String ausgibt. `pp` ("Pretty-Printing") versucht ein Object in bestmöglicher Darstellung auszugeben. Ist `as-code?` `#t`, dann wird Scheme-Code eingerückt ausgegen.

Eine detailiertere Beschreibung von `MIT Scheme` findet sich in dem oben zitierten Werk [HT91]. Dort ist der volle Sprachumfang, von dem wir hier nur einen kurzen Abriß gegeben haben, dokumentiert.

5 Gofer-Kurzübersicht

5.1 Funktionen und Operatoren

5.1.1 Funktionsnamen

Funktionen werden in Gofer 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 Gofer 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 Gofer sind:

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

5.1.2 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.

5.1.3 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.

5.1.4 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 Gofer (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 Gofer 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 /
```

5.2 Typen in Gofer

5.2.1 Typ–Namen

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

5.2.2 Der Typ-Operator (::)

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

42 :: Int   42 ist ein Integer-Wert

5.2.3 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

5.2.4 Standard-Typen in Gofer

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)

5.2.5 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.

5.2.6 Zeichen

Zeichenkonstanten werden durch Einschließen in Apostrophe dargestellt:

'a' :: Char

5.2.7 Zeichenketten

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

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

5.2.8 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)

5.2.9 Polymorphismus

In Gofer 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
```

5.2.10 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

5.2.11 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]
```

5.3 Funktionsdefinition

5.3.1 Pattern Matching

Funktionen werden in Gofer 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!`

5.3.2 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

5.3.3 Guarded Equations

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

```
f x1 x2 ... xn | bedingung1 = e1
               | bedingung2 = e2
                 ...
               | bedingungm = em
```

Beispiel:

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

5.3.4 Lokale Definitionen

Lokale Definitionen können in Gofer 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))

5.3.5 Unendliche Datenstrukturen

Lazy Evaluation erlaubt es in Gofer 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
```

5.3.6 Funktionen höherer Ordnung

In Gofer 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)
```

5.4 Spezielle Sprachkonstrukte

5.4.1 Arithmetische Folgen

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

`[n..]`	stellt die unendliche Liste dar, beginnend mit `n`, in aufsteigender Reihenfolge. `[2..] = [2, 3, 4, 5, ...`

`[n..m]`	stellt die Liste der Elemente von `n` bis `m` in aufsteigender Reihenfolge dar.
	`[-3..1] = [-3, -2, -1, 0, 1]`
	`verb|[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, ...`
	`[2,1..] = [2, 1, 0, -1, -2, ...`
	`[0,0..] = [0, 0, 0, 0, 0, ...`

`[n,k..m]`	stellt die Liste von `n` bis `m` dar, wobei die Schrittweite durch die Differenz aus `k` und `n` bestimmt wird.
	`[1,3..12] = [1, 3, 5, 7, 9, 11]`
	`[0,0..10] = [ 0, 0, 0, 0, ...`

5.4.2 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], even x ] = [ 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]
```

5.4.3 Lambda Ausdrücke

Lambda Ausdrücke gestatten die Definition namenloser Funktionen:

\ *atomicPatterns* -> *expr*

Beispiele für Lambda Ausdrücke:

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

5.4.4 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
```

5.4.5 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 Goferausdruck, 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.

5.4.6 Kommentare

Es gibt zwei Möglichkeiten in Gofer 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 −}

5.5 Layout rules - Einrücken von Gofercode

Gofer 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 Gofer 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 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).

5.6 Ein- und Ausgabe in Gofer

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. Reines funktionales Programmieren ist aber auf das Programmieren ohne Seiteneffekte ausgelegt.
Unsere Vorstellung von Input/Output Operationen widerspricht so dem funktionalen Paradigma im Wesentlichen an zwei Punkten.

- **referentielle Transparenz**
  I/O kann als ein Kommunikationsprozess zwischen einem Programm und dem Betriebssystem betrachtet werden. Das Programm schickt so zB mit einem *readfile* Befehl eine Botschaft an das Betriebssystem, die je nach Status des Filesystems mit einem bestimmten String oder aber einer Fehlermeldung vom System beantwortet wird. Betrachten wir den *readfile* Befehl als eine Funktion, die als Wert die Antwort des Betriebssystems hat, so verletzt diese Funktion die Forderung nach referentieller Transparenz. Je nach dem Status des Filesystems kann derselbe Funktionsaufruf `(readfile datei)` verschiedene Werte bekommen, abhängig davon, ob die Datei existiert, gelesen werden darf, gelesen werden kann und welcher Text unter *datei* abgespeichert ist. Diese sind alles dem Programm unbekannte Zustände.

- **Auswertungsreihenfolge**
  Als zweite Schwierigkeit stellt sich die Frage der Auswertungsreihenfolge: Lazy funktionale Sprachen sind gerade so konzipiert, daß die Auswertungsreihenfolge der einzelnen Redexe im Programmablauf streng nach der Notwendigkeit gesteuert wird. Erst wenn der Wert eines Ausdrucks in Normalform benötigt wird, wird dieser auch ausgewertet. Bei I/O Funktionen im klassischen Sinn interessiert weniger der Wert der Funktion, sondern der durch sie hervorgerufene Seiteneffekt. Es kann dem Programmierer schon mitunter entgehen, daß der Wert eines *write* Befehls zur Programmevaluierung nicht benötigt wird, und so dieser auch nicht ausgeführt wird.
  Ein Codeoptimierer könnte unter Umständen eine zweimalige Abfrage an den Benutzer wegoptimieren, da sie über denselben Ausdruck, der aber verschiedene Werte erwartet, realisiert wird.
  Im allgemeinen haben wir von I/O Prozessen die Vorstellung einer Sequenz einzelner I/O Befehle. Lazy funktionale Sprachen wollen aber Sequenzialisierung möglichst nicht erzwingen. So wird auch kein expliziter Sequenzoperator wie in imperativen Sprachen (C, Pascal) aber auch in strikten funktionalen Sprachen (Scheme) zur Verfügung gestellt.

Wir können von lazy funktionale Sprachen also nicht mit dem uns vertrauten I/O Routinen rechnen.
Betrachten wir nun verschiedene Lösungen des I/O Problems in Gofer/Haskell.

5.6.1 Dialogues

Die elementarste Art von Ein- und Ausgabe in Gofer stellen die Dialogfunktionen dar. Leider ist diese Art des I/O recht unhandlich und konterintuitiv. I/O Funktionen werden als ein Dialog zwischen Programm und Betriebssystem modelliert. [O'D85] Dieser Dialog stellt sich dar als eine unendliche Liste von Anfragen an das Betriebssystem `[Request]` auf Seite des Programms als Kommunikationspartner und eine ebensolche Liste von Antwortmeldungen auf diese Anfragen des Betriebssystems `[Response]`.
Wir haben also die zwei Listen:
$[req_1, req_2, req_3, ...]$
$[res_1, res_2, res_3, ...]$
als Teile unseres Kommunikationsprozesses, wobei jeweils res_i die Antwort des Betriebssystems auf die Meldung req_i des Programms sein sollen.
Gofer-Funktionen, die Daten einlesen oder ausgeben, müssen vom Typ `Dialogue` sein. `Dialogue` ist durch folgendes Typ-Synonym definiert:

```
type dialogue = [Response] -> [Request]
```

Eine Funktion dieses Typs erzeugt also eine Liste von Requests an das Betriebssystem und erhält von diesem die Antworten (Responses) als Eingabe. Mögliche Responses sind:

```
Success
Str String
Failure IOError
```

wobei `IOError` eine der folgenden Formen annehmen kann:

```
WriteError String
ReadError String
SearchError String
```

```
        FormatError String
        OtherError String
```

Mögliche Requests sind:

- `ReadFile` *string*: Liest das File mit Namen *string*.
  Mögliche Antworten sind:
  - `Str` *inhalt*, falls das Lesen erfolgreich war. *inhalt* ist ein String, der den Inhalt des Files enthält.
  - `Failure (SearchError` *name*`)`, falls der Name (*name*) des Files unbekannt ist
  - `Failure (ReadError` *name*`)`, falls ein anderer Fehler beim Lesen des Files *name* auftritt
- `WriteFile` *name string*: Schreibt die Zeichenkette *string* in das File *name* (ein nicht-existierendes File wird erzeugt, ein existierendes wird überschrieben). Mögliche Responses sind:
  - `Success`, falls das Schreiben des Files erfolgreich war.
  - `Failure (WriteError` *msg*`)`, falls das Schreiben nicht erfolgreich war. Trat der Fehler während des Öffnens des Files auf, so enthält *msg* den Namen des Files, sonst eine (druckbare) Fehlermeldung.
- `AppendFile` *name string*: Analog zu `WriteFile` mit dem Unterschied, daß *string* an ein existiernedes File *name* angehängt wird.
- `ReadChan` *name*: Liest den Inputstream *name* (z.B. `stdin`). Achtung: Es ist ein Fehler, mehr als einmal einen Inputstream zu lesen. Mögliche Responses sind:
  - `Str` *contents*, falls der Request erfolgreich war. *contents* (eine Zeichenkette) enthält die Eingabe.
  - `Failure (SearchError` *name*`)`, falls ein unbekannter Inputstream angegeben wurde.
  - `Failure (ReadError` *name*`)`, falls Inputstream schon einmal gelesen wurde.
- `AppendChan` *name string*: gibt die Zeichenkette *string* auf dem Channel *name* aus (z.B stdout, stderr, stdecho). Mögliche responses sind:
  - `Success`, falls der Request erfolgreich war.
  - `Failure (SearchError` *name*`)`, falls der angegebene Channel unbekannt war.
  - `Failure (WriteError` *msg*`)`, falls ein Fehler während des Schreibens auftrat. *msg* enthält dann eine druckbare Fehlermeldung.

Beispiele:

```
helloWorld :: Dialogue
helloWorld resp = [AppendChan stdout "hello, World"]

showFile :: String -> Dialogue
showFile name ~(read:_) = [ReadFile name,AppendChan stdout result]
```

```
where result = case read of Str contents -> contents
Failure _ -> "Can't open " ++ name

moin ~(~Success: ~(~(Str zahl) : ~(~Success:[]))) =
          [AppendChan stdout "Geben Sie ein Zahl ein: ",
           ReadChan stdin,
           AppendChan stdout ("\n Das Quadrat von "
                            ++ (head (lines zahl))
                            ++ " ist: "
                            ++ ausgabe
                            ++ "!\n" )]
            where
             ausgabe = int2string ((string2int (head (lines zahl)))^2)
```

(Das Pattern (read:_) darf erst ausgewertet werden, wenn der Request (ReadFile) abgesetzt wurde, deshalb muß es lazy ausgewertet werden (s. Irrefutable Pattern)) [4]

5.6.2 Continuations

Eine zweite Möglichkeit I/O in refentiell transparenter Weise in Gofer zu realisieren, ist über *Continuations*. Hier steht der sequentialisierte Charakter von I/O-Funktionen im Vordergrund. Eine I/O-Funktion hat dabei immer eine Reihe von Argumenten nach folgendem Muster: zuerst folgen die Argumente, die für die I/O-Operation notwendig sind. Dieses kann z.B. der Dateiname einer zu lesenden Datei sein. Dann folgt als ein weiteres Argument eine I/O-Funktion, die ausgeführt werden soll, wenn die I/O-Operation vom Betriebssystem nicht erfolgreich abgeschlossen werden kann, und als ein letztes Argument folgt die I/O-Funktion, die ausgeführt werden soll, wenn die I/O-Operation vom Betriebssystem erfolgreich abgeschlossen werden konnte. In der Gofer-Prelude sind solche Continuation-I/O-Funktionen definiert. Mit ihnen läßt sich recht bequem I/O in Gofer realisieren.
Beispiele:

```
helloWord = appendChan stdout "hello worls\n" abort done

showFile name
 = readFile
     name
     (\_ -> appendChan stdout  ("Can't open " ++ name) abort done)
     (\result -> appendChan stdout result abort done)

moin = appendChan stdout "Geben Sie ein Zahl ein: " abort readzahl

readzahl = readChan stdin abort writequadrat


writequadrat zahl  =  appendChan stdout
                              ("\n Das Quadrat von
                            ++ (head (lines zahl))
```

[4] Wer dieses Programm oder ein ähnliches ausprobiert, wird sehen, in welcher Weise die lazy Auswertungsstrategie nicht ganz zu dem erwarteten Ergebnis führt.

```
                              ++ " ist: "++ ausgabe ++ "!\n" )
                        abort done
            where
             ausgabe = int2string ((str2int (head (lines zahl)))^2)

str2int x = str2intaux x 0
str2intaux "" a = a
str2intaux (n:ns) a = str2intaux ns ((10*a)+(ord n) - 48)
```

[5]

5.6.3 Monadisches I/O

Aus den Problemen, die viele Programmierer mit Dialogen hatten, wurde das monadische I/O-Konzept entwickelt. [PJW93] Es basiert auf einer Programmiermethode, die in geschickter Weise Daten implizit weiterreicht, den Monaden. [Wad90, Wad92] Eine Sequentialisierung kann in rein funktionalen Sprachen nur durch Funktionkomposition adequat erreicht werden. Haben wir zwei I/O Funktionen f_1 und f_2, die nacheinander ausgeführt werden sollen, so ist dieses am einfachsten zu erreichen, wenn f_1 einen Wert hat den f_2 als Eingabeargument zur Evaluierung benötigt. Wenn dieser Wert von f_1 dann in irgendeiner Weise noch die Umgebung (Filesystem etc.) darstellt, dann ist auch die referentielle Transparenz gewährleistet.
Diesem Gedanken liegt das monadische I/O zugrunde. Monaden stellen zwei Operatoren zur Verfügung: `bind` ein Sequenzoperator, der durch Komposition realisiert wird, und `unit` einen Operator, der es erlaubt, komplexe Datentypen in eine Monade zusammenzupacken, die dann implizit weitergereicht werden. Es kann also gut die für I/O benötigte Umgebung (die `World`) in einer Monade enthalten sein und weitergereicht werden.
Betrachten wir die beiden Aspekte des monadischen I/O getrennt:

- **Sequenzialisierung**
  Wir benötigen zunächst einmal einen Operator, der uns aus einem einfachen Datentyp ein für I/O Operationen monadisch verpackten Datentyp generiert:
  `unitIO :: a -> IO a`
  Als zweiten Operator benötigen wir eine Funktion, die es uns erlaubt, Sequenzen von I/O Operationen mit dem monadisch verpackten Datentyp zu definieren:
  `bindIO :: IO a -> (a -> IO b) -> IO b`
  Das erste Argument wird hier das Ergebnis einer I/O Operation oder aber der unitIO-Funktion sein und das zweite Argument eine I/O Operation. `bindIO` soll nun die Komposition eines Ausdrucks, der einen I/O Seiteneffekt und als Ergebnistyp `IO a` hat mit einer weiteren I/O Operation darstellen.
  Mit diesen beiden Funktionen und vordefinierten I/O Funktionen kann nun in natürlicher Weise eine Funktion definiert werden, die ein Wort aus der Tastatur einliest:

  ```
  wort :: IO String
  wort = wortaux []

  wortaux String -> IO String
  ```

[5] Man sieht wieder, daß trotz des stark sequentialisierten Charakters der Continuation-Funktionen, das Programm `moin` nicht ganz die gewünschte Erscheinung hat.

```
wortaux w = getcIO `bindIO` \a ->
             if ((a== eof) \/ (a==' ')) then
                unitIO (reverse w)
             else
                (unitIO (a:w)) `bindIO` wortaux
```

- **referentielle Transparenz**
  Bisher haben wir noch nicht betrachtet, in welcher Weise der Datentyp `IO a` definiert ist, und wie die beiden Monadenfunktionen auf ihm operieren. Erst wenn wir jedoch genau wissen, wie die Umgebung in dieses System integriert ist, können wir erkennen, daß die referentielle Transparenz auch wieder durch *enviroment passing* für die I/O Funktionen gewährleistet ist.
  Der Typ `IO a` soll uns die Umgebung mit einem Datum des Typs a verbinden, so daß wir als versteckten Parameter auch die `World` in den Funktionen haben:

  ```
  type IO a = World -> IORes a
  data IORes a = MkIORes a World
  ```

  Der entscheidene Trick ist also, daß `IO a` eine Funktion ist, die für eine Umgebung ein Element von Typ `a` mit dieser Umgebung über den Konstruktor `MKIORes` verknüpft. Die monadischen Funktionen werden definiert als:

  ```
  unitIO a w    = MkIORes a w
  bindIO m k w  = case (m w) of
                    MkIORes a w' -> k a w'
  ```

  Der Typ `World` ist ein abstrakter Typ mit genau einer auf ihm definierten Funktion, der Funktion `ccall`. Diese Funktion ist das einzige Sprachkonstrukt, daß Haskell zum Implementieren des monadischen I/O hinzugefügt wurde. Es erlaubt eine beliebige C-Funktion aufzurufen, welche dann die Kommunikation mit der Umgebung betreibt. Mit Hilfe dieser Funktion werden die I/O-Funktionen definiert. Betrachten wir hierzu ein Beispiel:

  ```
  putcIO a =\w ->
    case a of
      MkChar a# ->
        case (ccall# putchar a# w) of
         MkIORes# n# w' -> MkIORes () w'
  ```

Die monadischen I/O-Funktionen sind im derzeitigen Haskell-Standard noch nicht eingebaut. Die meisten Haskell-Implementierungen bieten monadisches I/O mitlerweile an. Monadisches I/O wird sicherlich in Haskell 1.3 aufgenommen. Eine Vorabvorschlag hierzu ist schon auf dem WWW erhältlich. (Adresse siehe weiter oben)

5.7 Überladen in Gofer

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 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
        [] == (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

[AS85] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* McGraw-Hill, New York, 1985.

[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.

[HPW+92] Paul Hudak [ed.], Simon L. Peyton Jones [ed.], Philip Wadler [ed.], Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell. A non-strict purely functional language. version 1.2, 1992.

[HT91] Chris Hanson and The MIT Team and a cast of thousands. MIT Scheme Reference Manual, Edition 1.1 for Scheme Release 7.1.3, 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT 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.

[O'D85] JT O'Donnell. Dialogues: a basis for constructing programming environments. In ACM, editor, *Proc ACM Symposium an Language Issues in Programming Enviroments*, pages 19–27, Seattle, 1985.

[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.

[PJW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings 20th Symposium on Principles of Programming Languages*. ACM, 1993.

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

[Ste90] Guy L. Steele. *Coomon Lisp*. Digital Press, 1990.

[Wad90] P. Wadler. Comprehending monads. In *Proceedings of Symposium on Lisp and Functional Programming*, pages 61–78, Nice, France, June 1990. ACM.

[Wad92] P. Wadler. The essence of functional programming. In *Proceedings 19th Symposium on Principles of Programming Languages*, pages 1–14, January 1992.