Einführung in das Arbeiten mit Gofer

In den Übungen zu dieser Einführungsvorlesung wird die funktionale Programmiersprache *Gofer* verwendet. Dieses Arbeitsblatt soll den Umgang mit Gofer in einem ersten Überblick nahebringen. Es ist empfehlenswert, die hier gegebenen Beispiele am Rechner nachzuvollziehen.

1. Berechnung von Ausdrücken

Eine funktionale Programmiersprache ist eine Programmiersprache, die es erlaubt, Ausdrücke wie mit einem Taschenrechner zu berechnen. Ein Ausdruck wird eingegeben, wie er auf dem Papier stehen würde. Die Auswertung des eingegebenen Ausdrucks wird durch Betätigung der Return-Taste veranlaßt.

Zunächst aber wird das Gofer-Programmiersystem durch den UNIX-Befehl `gofer` aufgerufen. Man erhält auf dem Bildschirm eine Ausgabe etwa in der folgenden Form:

```
> gofer
Gofer Version 2.28a  Copyright (c) Mark P Jones 1991-1993

Reading script.gs file "/usr/local/dist/DIR/gofer/etc/standard.prelude":

Gofer session for:
/usr/local/dist/DIR/gofer/etc/standard.prelude
Type :? for help
?
```

Nach Erscheinen des Fragezeichens ist der Benutzer aufgerufen, etwas einzugeben. An dieser Stelle ist es dem Benutzer erlaubt, nach dem Fragezeichen ("Gofer-Prompt") Ausdrücke zur Auswertung einzugeben und deren Auswertung schließlich zu veranlassen.

Einfache Beispiele sind arithmetische Ausdrücke wie etwa $3+4$. Mit Gofer sieht die Berechnung dieses Terms wie folgt aus:

```
? 3 + 4
7
(3 reductions, 6 cells)
```

Die erste Zeile enthält die Eingabe (abgeschlossen mit, hier nicht sichtbarem, Return). Das (korrekte) Ergebnis läßt sich von der zweiten Zeile (= erste Ausgabezeile) ablesen. Die dritte Zeile (=zweite Ausgabezeile) zeigt die Anzahl der vom Rechner ausgeführten Berechnungsschritte und die Anzahl der dabei benutzten Speichereinheiten.

Wie in der Mathematik üblich, können Klammern verwendet werden, um Teile eines komplizierteren Ausdrucks zusammenzufassen. Beispiel:

```
? 2 * (3 + 4)
14
(5 reductions, 9 cells)
```

Den Absolutbetrag einer Zahl, würde man in der Mathematik etwa $abs(x)$ schreiben. Ebenso funktioniert das in Gofer. Beispiel:

```
? abs (-5)
5
(9 reductions, 11 cells)
```

Um den Funktionsnamen (wie hier `abs`) vom nachfolgenden Argument klar unterscheiden zu können, verwendet man in Gofer normalerweise ein Leerzeichen, wie im Beispiel geschehen, denn Leerzeichen sind in Gofer nie Bestandteil von Bezeichnungen (Identifikatoren) wie insbesondere von Funktionsbezeichnungen (Funktionsidentifikator, Funktionsname). Ein Identifikator (Bezeichner) ist eine mit einem Buchstaben beginnende Folge von Zeichen, die ein Buchstabe, eine Ziffer oder das Zeichen Unterstrichen ("_") sein können. Mit jedem anderen Zeichen in einem Gofer-Programmtext endet der Identifikator. Ein korrekter Ausdruck ist also auch `abs(-5)`, weil offensichtlich der Funktionsidenfikator das Zeichen `(` nicht enthalten kann.

Wollen wir den Absolutbetrag von 5 berechnen, so können wir zum Beispiel `abs (5)` oder `abs 5` eingeben. `abs5` dagegen würde das Gofer-System als neuen, von `abs` zu unterscheidenden Funktionsbezeichner interpretieren.

```
? abs 5
5
(5 reductions, 11 cells)
```

Bei dem Ausdruck `abs (-5)` ist das Weglassen der Klammern nicht möglich. Nach Eingabe von `abs -5`würde Gofer `-` als zweistellige Subtraktion interpretieren. Die Meldung eines Typfehlers wäre die Folge, da abs nicht vom Typ "ganze Zahl" ist, sondern den Typ einer Funktion besitzt. Der Ausdruck `abs 3 - 5` ist aber korrekt und hat den Wert -2.

2. Kommando zur Beendigung der Gofer-Sitzung

Nach dem Start wartet das Gofer-Programmiersystem auf einen Ausdruck, wertet einen eingegebenen Ausdruck aus, gibt das Ergebnis aus und wartet auf die nächste Eingabe. Statt eines Ausdrucks können aber auch sogenannte *Kommandos* eingegeben werden. Alle Kommandos erkennt man daran, daß sie mit einem Doppelpunkt beginnen. Kommandos werden nicht wie Ausdrücke ausgewertet, sondern sie haben eine direkte Wirkung auf das Gofer-Programmiersystem. Um die Sitzung mit Gofer zu beenden, braucht man ein Kommando, und zwar `:q` wie folgt:

```
? :q
[Leaving Gofer]
>
```

3. Benutzerdefinierte Funktionen und weitere Gofer-Kommandos

Wie in der Mathematik ist in Gofer die Definition von Funktionen ein entscheidendes Strukturierungsmittel (z.B. zur Verkürzung der Aufschreibung von Ausdrücken). Dadurch erst wird es möglich, allgemeine Algorithmen in Gofer zu programmieren. Wenn zum Beispiel der mathematische Ausdruck $x^2 + 5x + 3$ für viele verschiedene x ausgewertet werden muß, dann möchte man sich eine Funktion $f(x) = x^2+5x+3$ definieren und diese dann auf die Werte anwenden. In der Gofer-Schreibweise lautet eine entsprechende Deklaration `f x = x^2 + 5*x + 3`. Sie entspricht der obigen Gleichung, durch die die Funktion f definiert ist. Durch diese Deklaration wird der Funktionsbezeichner f an die entsprechende Abbildungsvorschrift *gebunden*.

In Gofer werden (globale) Definitionen (für Funktionen) in einer gesonderten Datei, dem sogenannten *Skript* aufbewahrt. Ein solches Skript wird (a) mit einem Editor erstellt und (b)

in Gofer "eingelesen". Die eingelesenen Definitionen stehen dann dem Benutzer zur Verfügung, um in Ausdrücken verwendet zu werden.

Verfolgen wir das obige Beispiel. Zunächst soll (von UNIX aus) eine Datei `script.gs` erstellt werden, die nur die obige Definition enthält. Dazu kann man einen beliebigen Texteditor verwenden, zum Beispiel den Editor `vi` oder `emacs`. Nach Eingabe des UNIX-Kommandos `cat script.gs` wird die Datei wie folgt auf den Bildschirm ausgegeben:

```
> cat script.gs
f x = x^2 + 5*x + 3
```

Nach dem Edieren kann man das Skript im Gofer-Programmiersystem mit dem Kommando `:l script.gs` einlesen:

```
? :l script.gs
Reading script.gs file "script.gs":

Gofer session for:
/usr/local/dist/DIR/gofer/etc/standard.prelude
script.gs
```

Jetzt kann die Funktion `f` wie eine vorgegebene Funktion in Ausdrücken verwendet werden:

```
? f 6
69
(19 reductions, 31 cells)
```

In der Meldung "`Gofer session for: ...`" wird das Verzeichnis der aktuell eingelesenen Skripten dargestellt. Das Prelude-Skript `standard.prelude` enthält die Definition aller Standardfunktionen, wie etwa die Definition von `abs`, und ist immer in diesem Verzeichnis enthalten, d.h. es braucht nicht extra eingelesen zu werden.

4. Ein weiterführendes Beispiel

Eine aus der Mathematik bekannte Funktion ist die Fakultätsfunktion, die man oft mit einem nachgestellten Rufezeichen schreibt, also zum Beispiel $6! = 1*2*3*4*5*6 = 720$. Diese Funktion kann man nicht (so wie `f` im letzten Abschnitt) einfach aus arithmetischen Grundfunktionen zusammensetzen. Vielmehr muß man eine sogenannte *rekursive Definition* verwenden, also eine, in der die Berechnung des Funktionswertes für ein Argument schrittweise auf Berechnungen derselben Funktion mit einfacheren Argumenten zurückgeführt wird.

Ein rekursive Definition der Fakultät aus einem Mathematikbuch sieht zum Beispiel so aus:

$$n! = \begin{cases} 1 & \text{wenn } n = 0 \\ n * (n-1)! & \text{sonst} \end{cases}$$

Eine andere Schreibweise für diese Definition ist:

$$n! = \textbf{ wenn } n = 0 \textbf{ dann } 1 \textbf{ sonst } n * (n-1)!$$

Diese Definition ist schon ziemlich nahe an der Definition einer Gofer-Darstellung der Funktion `fac`. Wir müssen nur noch berücksichtigen, daß in der mathematischen Schreibweise das Zeichen $=$ in zwei verschiedenen Bedeutungen auftritt, und zwar als Definitionszeichen in $n! = \ldots$ und als Test auf Gleichheit in **wenn** $n = 0 \ldots$. Diese unterschiedlichen Bedeutungen werden in Gofer durch unterschiedliche Schreibweisen deutlich gemacht. Wir erhalten die folgende Gofer-Definition:

```
fac n  =  if n==0 then 1 else n * fac (n-1)
```

Wenn wir diese Definition in ein Skript schreiben und dieses laden (d.h., einlesen mittels des Lesekommandos `:l` *script.gs*) , steht uns die Fakultätsfunktion zur Verfügung:

```
? fac 6
720
(30 reductions, 45 cells)
```

Eine alternative Definition der Fakultät beruht auf dem Gedanken, eine Menge von Gleichungen anzugeben, die den Wert einer Funktionsanwendung für alle Argumente festlegen. Zum Beispiel könnte eine mathematische Definition wie folgt aussehen:

$$\begin{aligned} 0! &= 1 \\ n! &= n * (n-1)! \qquad \textbf{für } n > 0 \end{aligned}$$

Diese Schreibweise kann fast ohne Änderung in Gofer übertragen werden:

```
fac1 0  =  1
fac1 n  =  n * fac1 (n-1)
```

Beachten Sie, daß in Gofer die Bedingung $n > 0$ weggelassen wurde. Gofer benutzt nämlich bei der Funktionsauswertung stets die erste anwendbare Gleichung. Es ist klar, daß bei Benutzung der zweiten Gleichung die Bedingung $n < 0$ dann sowieso erfüllt ist. In Gofer ist also die Reihenfolge der Aufschreibung wichtig. Deshalb würde eine Vertauschung der beiden Gleichungen in der Definition von `fac1` zur Folge haben, daß ausschließlich die erste Gleichung zur Anwendung käme, weshalb dieses Programm dann nie terminieren würde.

Die Funktion `fac1` liefert die gleichen Ergebnisse wie `fac`:

```
? fac1 6
720
(20 reductions, 31 cells)
```

Die Programme `fac` und `fac1` sind Beispiele für verschiedene funktionale Programmierstile (bedingte Ausdrücke kontra definierende Gleichungen).

Ein Skript enthält im allgemeinen mehrere Funktionsdefinitionen. Diese sind dann durch Leerzeilen getrennt und dürfen sich wechselseitig aufrufen.

5. Bestandteile von Ausdrücken

Ausdrücke enthalten stets Bezeichnungen für Operationen bzw. Funktionen, Objekten, oder Variablen für Objekte. Beispielsweise werden in dem Ausdruck `3+abs(x)` die Additionsoperation mit `+`, die Zahl 3 mit dem Zeichen `'3'`, die Betragsfunktion mit `abs` und die Variable für Zahlen mit `x` bezeichnet. Eine bestimmte Menge von Operationen bzw. Funktionen und Objekten, zusammen mit deren Bezeichnungen werden Rechenstrukturen genannt. Damit stellt sich die Frage, welche Rechenstrukturen von Gofer verstanden, d.h. interpretiert werden können. Diese Frage wird in den nächsten Abschnitten behandelt.

Die Rechenstruktur der Wahrheitswerte

Funktionen mit der Funktionalität $\rightarrow$ `Bool`*:*

`True`	„wahr“ (**L**)	`False`	„falsch“ (**O**)

Funktion mit der Funktionalität `Bool` $\to$ `Bool`*:*

`not`	logische Negation ($\neg$)

Operationen mit der Funktionalität `Bool` $\times$ `Bool` $\to$ `Bool`*:*

`==`	gleich , strikte Äquivalenz ($\Leftrightarrow$)	`&&`	sequentielle Konjunktion ($\wedge$)
`||`	sequentielle Disjunktion ($\vee$)		

Die Rechenstruktur der ganzen Zahlen

Funktionen mit der Funktionalität $\to$ `Int`*:*

Hierunter sind alle Ziffernfolgen (kein Vorzeichen!) zu verstehen, insbesondere `0` für die Null.

Funktionen bzw. Operationen mit der Funktionalität `Int` $\to$ `Int`*:*

`negate`	einstelliges Minus ($-$)	`abs`	Absolutbetrag ($\|\cdots\|$)
`-`	äquivalent zu `negate`	`signum`	Vorzeichenfunktion (sgn)

Operationen mit der Funktionalität `Int` $\times$ `Int` $\to$ `Int`*:*

`+`	Addition	`-`	Subtraktion
`*`	Multiplikation	`/`	ganzzahlige Division ($\div$)
`^`	Exponentiation, „hoch“		

Funktionen mit der Funktionalität `Int` $\to$ `Bool`*:*

`even`	Test auf gerade Zahl	`odd`	Test auf ungerade Zahl

Operationen mit der Funktionalität `Int` $\times$ `Int` $\to$ `Bool`*:*

`==`	gleich	`<=`	kleiner-gleich ($\leq$)
`/=`	ungleich	`>=`	größer-gleich ($\geq$)
`<`	kleiner	`>`	größer

Die Rechenstruktur der Zeichen

Operationen mit der Funktionalität $\to$ `Char`*:*

Hierunter sind in Apostrophen eingeschlossene Einzelzeichen `'`c`'`, wobei c für ein von der Tastatur einzugebendes Zeichen steht, zu verstehen. TAB und RETURN werden ausnahmsweise durch `'\t'` bzw. `'\n'` dargestellt.

Funktion mit der Funktionalität `Char` $\to$ `Int`*:*

`ord`	Ordnung, numerische Zeichencodierung (ASCII-Code)

Funktion mit der Funktionalität `Int` $\to$ `Char`*:*

`chr`	Dekodiertes Zeichen (ASCII-Code), z.B. ergibt `chr 65` das Zeichen `'A'`

Operationen mit der Funktionalität `Char` $\times$ `Char` $\to$ `Bool`*:*

`==`	gleich	`<=`	kleiner-gleich ($\leq$)
`/=`	ungleich	`>=`	größer-gleich ($\geq$)
`<`	kleiner	`>`	größer

Die Rechenstruktur der Listen (Sequenzen)

Sei `t` ein beliebiger Typ (definierte Menge von Objekten). Dann ist `[t]` der Typ der Sequenzen von Objekten des Typs `t`.

Operationen mit der Funktionalität $\rightarrow$ `[t]`*:*

Hierunter verstehen wir die Listenkonstruktoren (Aufzählungsoperationen):

`[]`	leere Liste (ε)
`[`a`]`	ein-elementige Liste mit dem Element a vom Typ `t`
`[`$a_1, \ldots, a_n$`]`	n-elementige Liste mit den Elementen $a_1, \ldots, a_n$ vom Typ `t`

Operation mit der Funktionalität `t` $\times$ `[t]` $\rightarrow$ `[t]`*:*

`:` Anfügen eines Elements: $a\!:\!l$ ergibt $append(a, l)$

Funktionen mit der Funktionalität `[t]` $\rightarrow$ `t`*:*

`head`	erstes Element	`last`	letztes Element

Funktionen mit der Funktionalität `[t]` $\rightarrow$ `[t]`*:*

`tail`	Rest der Liste nach Entfernen des ersten Elements
`init`	Rest der Liste nach Entfernen des letzten Elements

Operation mit der Funktionalität `[t]` $\times$ `[t]` $\rightarrow$ `[t]`*:*

`++` Konkatenation ($\circ$ bzw. conc)

Funktionen mit der Funktionalität `t` $\times$ `[t]` $\rightarrow$ `Bool`*:*

`elem`	ist enthalten?	`notElem`	ist nicht enthalten?

Operationen mit der Funktionalität `[t]` $\times$ `[t]` $\rightarrow$ `Bool`*:*

`==`	gleich	`<=`	kleiner-gleich — nach lexikographischer Ordnung ($\leq$)
`/=`	ungleich	`>=`	größer-gleich ($\geq$)
`<`	kleiner	`>`	größer

Funktion mit der Funktionalität `[t]` $\rightarrow$ `Int`*:*

`length` Länge einer Liste

Operation mit der Funktionalität `[t]` $\times$ `Int` $\rightarrow$ `t`*:*

`!!` Indexzugriff: l `!!` i ergibt das i-te Element der Liste l

Funktionen mit der Funktionalität `Int` $\times$ `[t]` $\rightarrow$ `[t]`*:*

`take`	Listenverkürzung (Iteration von head): `take i l` ergibt $[l_1, \ldots, l_i]$
`drop`	Listenverkürzung (Iteration von tail): `drop i l` ergibt $[l_{i+1}, l_{i+2}, \ldots]$

Die Rechenstruktur der Zeichenreihen

Der Typ `String` ist synonym zu `[Char]`. Deshalb gelten alle Listenoperationen entsprechend. Hier sei nur die spezielle Notation für Zeichenreihen in Gofer aufgeführt.

Operationen mit der Funktionalität $\rightarrow$ `String`*:*

Anstelle von `['`c_1`',...,'`c_n`']` ist es zulässig, die Notation `"`$c_1 \cdots c_n$`"` zu verwenden. Außerdem wird die leere Zeichenreihe (ε) auch durch `""`, anstelle von `[]`, dargestellt.

Die Rechenstruktur der Tupel

Seien $\mathtt{t}_1$, ..., $\mathtt{t}_n$ beliebige, nicht notwendig verschiedene Typen.

Typen: `()`, $(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ $(n > 1)$, $\mathtt{t}_1$, ..., $\mathtt{t}_n$; `Bool`

Operationen mit der Funktionalität $\rightarrow$ `()` *bzw.* $\rightarrow (\mathtt{t}_1, \ldots, \mathtt{t}_n)$*:*

Hierunter verstehen wir die Tupelkonstruktoren:

`()`	leeres Tupel (einziges Element des Typs `()`)
$(a_1, \ldots, a_n)$	n-Tupel mit den Komponenten $a_1, \ldots, a_n$, wobei für jedes i $(1 \leq i \leq n)$ Komponente a_i vom Typ $\mathtt{t}_i$ ist.

Hinweis: Für den ausgeschlossenen Fall $n = 1$ ist der Ausdruck (a) identisch mit a (geklammerter Ausdruck); dementsprechend wird auch der Typ$(\mathtt{t}_1)$ mit $\mathtt{t}_1$ gleichgesetzt („geklammerter Typenausdruck“).

Die weiteren Funktionen sind nur für Paare eingeführt.

Funktionen mit der Funktionalität $(\mathtt{t}_1, \mathtt{t}_2) \rightarrow \mathtt{t}_1$ *bzw.* $(\mathtt{t}_1, \mathtt{t}_2) \rightarrow \mathtt{t}_2$*:*

`fst`	Projektion auf die erste Komponente (`fst` (a, b) ergibt a)
`snd`	Projektion auf die zweite Komponente (`snd` (a, b) ergibt b)

Operationen mit der Funktionalität $(\mathtt{t}_1, \mathtt{t}_2) \times (\mathtt{t}_1, \mathtt{t}_2) \rightarrow$ `Bool`*:*

`==`	gleich	`<=`	kleiner-gleich — nach lexikographischer Ordnung $(\leq)$
`/=`	ungleich	`>=`	größer-gleich $(\geq)$
`<`	kleiner	`>`	größer

Gewöhnlich werden Tupel bei Mehrfachergebnissen eingesetzt. Als Beispiel betrachten wir die folgende, in Gofer standardmäßig vorhandene

Funktion mit der Funktionalität `Int` $\times$ `[t]` $\rightarrow$ `([t],[t])`*:*

`splitAt`	Listenaufspaltung (Zusammenfassung von `take` und `drop`; `splitAt i l` ergibt das Paar $([l_1, \ldots, l_i], [l_{i+1}, l_{i+2}, \ldots])$

*

*   *

Kurze Hinweise zu Gofer erhalten Sie von Unix aus mit dem Unix-Kommando `man gofer`.