Technische Universität München
Institut für Informatik
Prof. Dr. M. Broy

Einführung in die Informatik I
WS 2001/2002
Arbeitsblatt 3
29. 11. 2001

Einführung in Gofer (2)

Dieses Arbeitsblatt verfolgt das Ziel, die Einführung in Gofer zu fundieren insofern es die Syntax korrekter Gofer-Programme und deren Aufruf darstellt. Ein *Gofer-Programm* ist nichts anderes als eine Folge von Deklarationen (Definitionen), die in einer Datei (wir nennen diese Datei hier stets *script*) abgelegt und in das Gofer-System mit dem load-Kommando "geladen" werden kann. Ein *Programmaufruf* ist nichts anderes als ein Term (Ausdruck), in dem die irgendwo schon definierten Funktionsymbole als Bezeichner vorkommen dürfen und der nach dem Prompt des Gofer-Systems eingegeben wird.

1. Allgemeines

Das Gofer-System stellt zunächst eine Menge von primitiven, d.h. fest implementierten Rechenfunktionen zur Verfügung, wie z.B. die arithmetischen Grundoperationen $(+, *\ldots)$, sowie die dafür erforderlichen Standardtypen, d.h. Mengen von Werten (z.B. ganze Zahlen) zusammen mit den entsprechenden Sortensymbolen (z.B. `Int`), auf denen die Rechenfunktionen operieren. Weitere nützliche Rechenfunktionen sind mittels der stets geladenen Datei standard.prelude vordefiniert. Diese Rechenfunktionen sind auf die primitiven Rechenfunktionen und Standardtypen abgestützt. Durch die Gesamtheit aller so vordefinierten Funktionen und Wertemengen, zusammen mit deren Bezeichnungen (Funktionssymbolen und Sorten), ist durch Gofer eine komplexe Rechenstruktur gegeben.

Die Signatur dieser Rechenstruktur ist durch die vordefinierten Bezeichnungen (Symbole) für Funktionen und Standardwertemengen gegeben. Die Signatur stellt die syntaktische Basis der Termsprache aller korrekten Goferausdrücke, mithin auch von Programmaufrufen, dar. Die Semantik dieser Rechenstruktur ist durch den Begriff des Wertes gegeben, der den Funktionssymbolen und den Sorten und insbesondere den Goferausdrücken zugeordnet ist. Den Sorten sind dabei Wertemengen, den Funktionssymbolen Funktionen zugeordnet.

Der Benutzer kann nun durch eigene *Funktionsdeklarationen* diese vorgegebene Rechenstruktur beliebig erweitern. Er deklariert einfach in einer Datei *script* neue Rechenfunktionen auf der Basis der bereits vorhandenen Rechenfunktionen und lädt *script* ins System. Dadurch steht nun eine erweiterte Rechenstruktur zur Verfügung mit einer erweiterten Signatur (inklusive der neuen Funktionsbezeichnungen) und einer entsprechend erweiterteten Sprache für Ausdrücke (Terme). Durch eine *Datendeklaration* (ebenfalls in script) kann auch eine Erweiterung der Signatur hinsichtlich der Typen (d.h. der Sorten und deren Wertemengen) definiert werden, die wiederum beliebig als Urbildbereiche und Bildbereiche neuer Funktionen zur Verfügung stehen. Die bereits vordefinierte Tupel- und Sequenzbildung (Typ (A,B) und Typ [A]) beliebiger schon definierter Typen (A und B) fällt in diese Systematik.

Besonders wichtig ist, daß Gofer für schon definierte Typen A und B auch den Typ A→B der Funktionen von A in B als Typ zuläßt. Damit sind Rechenstrukturen mit Funktionen sogenannten "höheren" Typs zugelassen. Gofer basiert sogar wesentlich auf dem Konzept der Funktionen höheren Typs und gewinnt dadurch eine große Flexibilität und konzeptionelle Einfachheit.

Zentraler Begriff in Gofer sind *Ausdrücke* und deren *Auswertung*. Ausdrücke bestehen i.a. aus einer Folge von Funktionsanwendungen (Applikationen). Die Auswertung von Ausdrücken wird auf die Auswertung von Rechenfunktionen (genauer gesagt, Rechenfunktionsanwendungen, wie z.B. 3+5) zurückgeführt. Es gibt in Gofer drei Arten der Auswertung von Rechenfunktionen.

1. Primitive Rechenfunktionen (z.B. +, Addition ganzer Zahlen) werden mit Maschinencode ausgewertet.

2. Gleichungsdefinierte Rechenfunktionen werden durch Termersetzung ausgewertet. Die Gleichungen hierfür sind in den Funktionsdeklarationen zu finden.

3. Spezielle Rechenfunktionen, die sogenannten Konstruktoren, werden trivial ausgewertet: Jeder Term, der nur aus Konstruktorsymbolen aufgebaut ist, wird sich selbst bzw. einer eineindeutigen Kodierung desselben zugeordnet. Benutzerdefinierte Konstruktoren werden zusammen mit den zugehörigen neuen Werttypen in Datendeklarationen definiert.

Wir halten fest, daß in Gofer Rechenfunktionen als *Konstruktoren* ausgezeichnet werden können. Konstruktoren werden innerhalb von Datenklarationen definiert. Wichtiges Kennzeichen der Deklaration von Konstruktoren in Gofer ist, daß für diese natürlich keine Gleichungen definiert werden können. Die Semantik dieser Konstruktoren besteht in speziellen bijektiven Funktionen, die die Bezeichnung von Werten eineindeutig mit ihrem Wert verknüpft. Im Prinzip sind Konstantenbezeichnungen z.B. von ganzen Zahlen (z.B. 103), oder Listen `(1:(2:(5:[])` von dieser Art. Später werden wir lernen, daß Gofer mit den Datendeklarationen nichts anderes gestattet als eine Erweiterung seiner gegebenen Rechenstruktur mit einer beliebigen Signatur und der zugehörigen "Termalgebra". Unter dem Stichwort der Termalgebra kann dann das gesamte Thema "rekursiver Datenstrukturen" in Gofer abgehandelt werden.

Die Deklaration von primitiven Rechenfunktionen werden wir nicht weiter betrachten. Hierzu ist eine detaillierte Kenntnis des standard.prelude erforderlich. Wir halten lediglich fest, daß in Gofer auch primitive Rechenfunktionen deklariert werden können.

Zunächst werden in den Übungen die Funktionsdeklarationen behandelt werden. Funktionsdeklarationen sind nichts anderes als (ev. bedingte) Gleichungen (eventuell mit sogenannter where-Klausel, die lediglich lokale, d.h. nur für den speziellen Ausdruck geltende Bezeichnungen vereinbart). Eine Menge solcher Gleichungen, eventuell ergänzt durch eine Erklärung der Funktionalität der Funktion in Form einer Typnebenbedingung (z.B. `f::Int->Int`), definiert eine Rechenfunktion. Wir sprechen dann von einer gleichungsdefinierten Funktion.

Bezeichner von Funktionen, insbesondere von Konstruktorfunktionen sind in Gofer entweder eine mit einem Buchstaben beginnende Folge von Buchstaben, Ziffern, dem Unterstrich (_) oder Apostrophen ('). Diese Bezeichner (Identifier) bestimmen eine applikative Schreibweise in den Ausdrücken (z.B. f x) . Bezeichnet ein Identifier eine Konstruktorfunktion, dann muß das erste Zeichen ein Großbuchstabe sein. Oder der Bezeichner besteht aus einer Folge von Sonderzeichen `@,$,%,&,*,+, -, . ,/ , :,<,=,>,?,#,\,^,`,|,~`. Dann nennt man den Bezeichner einen *Operator*. Die Operatoranwendung in Ausdrücken folgt einer Infixschreibweise, oder Varianten davon. Operatoren die einen Konstruktor bezeichnen sind genau diejenigen, die mit einem Doppelpunkt beginnen.

2. Syntax eines Gofer-Skripts

Ein Gofer-Skript ist eine Folge von durch Semikolon (;) getrennte Deklarationen. Die Folge ist in geschweiften Klammern ({,}) eingeschlossen. Es gibt

- Funktionsdeklarationen

- Datendeklarationen

- Typdeklarationen

- Sonstige: primitive Funktionen, usw.

Es gibt zwei Formen der Funktionsdeklaration, nämlich Gleichungen und die Definition des Typs (Funktionalität) für Funktionen.

Beispiel:

`{f::Bool->Bool}`	Funktionalität von `f`
`{f x | x==True  =False}`	bedingte Gleichung für `f`
`{f::Bool->Bool; f x | x==True  =False}`	Folge von 2 Deklarationen.

Die Form der Gleichungen in Funktionsdeklarationen werden wir in einem nachfolgenden Abschnitt genauer betrachten, ebenso wie die Daten- und Typdeklarationen. Zunächst sind die folgenden Regeln zu merken.

Regeln:

1. Falls ein Skript als Datei geschrieben wird und über das load-Kommando in das Gofersystem eingelesen wird, dann entfallen die äußeren geschweiften Klammern.
2. Gleichungen für ein und dieselben Funktionsbezeichnungen müssen stets unmittelbar aufeinander folgen. (Leerzeilen spielen keine Rolle)
3. Das Semikolon und die geschweiften Klammern sind gleichbedeutend mit bestimmten Formatierungen durch Zeilenumbrüche und Einrückungen (Abseitsregel)

Wir betrachten die Wirkungsweise von Formatierungen in dem folgenden Beispiel einer Funktionsdeklaration.

```
f::Bool->Bool
f x | x==True   =False
    | otherwise = True
```

Die Formatierung liest Gofer so, daß nach der ersten Zeile ein Semikolon ergänzt wird, nicht aber nach der zweiten Zeile, d.h. die dritte Zeile wird einfach als Fortsetzung der zweiten Zeile interpretiert.

Die Verletzung dieser Regeln liefert oft "unerklärliche" Fehler.

```
f::Bool->Bool;
  f x | x==True   =False
 | otherwise = True
```

Das Zeichen `|` der dritten Zeile ist gegenüber der vorausgegangen Zeile "im Abseits", d.h. es wird ein Semikolon eingefügt. Dies ist aber syntaktisch falsch. Fehlermeldung!

3. Syntax eines Goferprogramms

Ein Goferprogramm ist ein `let`-Ausdruck der Form

`let` *script* `in` *Ausdruck*

Der Ausdruck wird dem Gofersystem nach dem Prompt eingegeben, die Ausführung wird mit `Return` veranlaßt. Zum Beispiel

```
let {x::Bool; x=True} in x&&False
```

Ein `let`-Ausdruck kann an jeder Stelle geschrieben werden, an der ein Ausdruck gefordert ist, z.B. auf der rechten Seite einer Gleichung:

```
y :: Bool
y = let {x::Bool; x=True} in x&&False
```

oder

```
x :: Bool
x = let {x::Bool; x=True} in x&&False
```

Man beachte den "Gültigkeitsbereich" der Bezeichnungen `x`. Wir halten fest, daß das Gofersystem nichts anderes macht als die Auswertung von Ausdrücken.

4. Gofer-Ausdrücke

Ein Gofer-Ausdruck ist entweder ein

1. einfacher Ausdruck,
2. eine Funktionapplikation,
3. eine Operatoranwendung,
4. ein bedingter Ausdruck (if-then-else)
5. ein `let`-Ausdruck
6. ein sonstiger Ausdruck, wie case-, Lambda usw.

Jeder Ausdruck besitzt den Typ seines Ergebnisses. Jede Ersetzung einer Variablen eines Typs A in einem Ausdruck durch einen Ausdruck vom Typ A ergibt wieder einen Ausdruck (korrekte Substitution, Termersetzung).

Zu den einfachen Ausdrücken gehören die Variablen, die Konstanten, die geklammerten Ausdrücke, die Tupel und die Listen, z.B.

```
x, 13, 'a', "abc", (x+5), (1,7), []
```

Die Funktionsapplikation besteht aus dem Funktionsbezeichner gefolgt von einem Argument (eventuell iteriert). Man beachte, daß etwa in dem Ausdruck `f (x,y)` dem Funktionsbezeichner `f` nur ein Argument folgt, nämlich das Tupel `(x,y)`. Der Ausdruck `f x y` bedeutet eine iterierte Funktionsanwendung. `f` wird dabei zunächst auf `x` angewandt und ergibt eine Funktion. Diese Funktion wird dann auf `y` angewandt. Einfache Beispiele sind

```
f x, f (x), f (x,y), f x y.
```

Die Operatoranwendung ist die Infixschreibweise einer Operation, oder Varianten der Infixschreibweise. Man kann diese Schreibweise in einem Skript individuell deklarieren. Einfache Beispiele sind

```
x + 3, x&&y
```

Die bedingten Ausdrücke haben die Form

`if` *Ausdruck* `then` *Ausdruck* `else` *Ausdruck*

wobei der Typ des ersten Ausdrucks `Bool` ist. Der Typ des zweiten Ausdrucks muß gleich sein dem Typ des dritten Ausdrucks. Beispiel:

```
if x==0 then 1 else 0
```

Die Form des `let-` Ausdrucks wurde bereits im Abschnitt Goferprogramme beschrieben. Die Semantik der `let-` Ausdrücke knüpft an mathematische Sprechweisen an, z.B. "Sei x=5 in dem Ausdruck x+3". Dann ist klar, daß der Ausdruck 5+3 gemeint ist. Die Bezeichnung x hat nur Gültigkeit in dem Ausdruck x+5, nicht aber außerhalb des Ausdrucks. In Gofer liest sich das gleiche wie folgt:

```
let x=5 in x+3
```

Die sonstigen Ausdrücke werden wir vorerst nicht benötigen und deshalb erst später betrachten.

5. Muster

In Gofer gibt es spezielle Ausdrücke, die auch *Muster* genannt werden. Gemeint sind Ausdrücke, die ausschließlich aus Konstruktoren und (ungebundenen) Variablen aufgebaut sind (Konstruktorterme mit Variablen). Auch Konstantenbezeichnungen (Konstruktorterme ohne Variablen) zählen dazu. Beispielsweise

```
17,  x:ys, (x,1,y), F x y, F (1:xs) (F x 3),
```

Daß `F` hier einen Konstruktor bezeichnet, sieht man an der Großschreibung. Die kleingeschriebenen Variablen dürfen natürlich im Skript nicht als gleichungsdefinierte Funktionen vereinbart worden sein.

Muster treten in Funktionsdeklarationen, z.B. auf der linken Seite einer Gleichung als Argumente von Funktionsbezeichnern auf.

```
f 1 (x:xs) = (1,xs)
```

Muster dürfen auch auf der Ergebnisseite von Funktionen auftreten, z.B. innerhalb der Deklarationen in `let`-Ausdrücken.

```
g s = let  (u,v)=f 1 s   in  v
```

Der Aufruf `g [4,2,3]` hat dann das Ergebnis `[2,3]`. Bei der Berechnung wird der Ausdruck `f 1 (x:xs)` gegen den Ausdruck `f 1 [4,2,3]` "gematcht", was zu einer eindeutigen Variablenbindung `x=4` und `xs=[2,3]` führt mit dem Ergebnis `(u,v)=(1,[2,3])`. Die Variable `v` wird dann an `[2,3]` gebunden und ausgegeben.

6. Typausdrücke

Die Deklaration des Typs von Funktionen benutzt bereits die Möglichkeit in Gofer aus einfachen, schon definierten Typen komplexere Funktionstypen zu konstruieren. So ist `Bool->Bool` ein Wertetyp, der sicherlich nicht zu den Standardtypen zu rechnen ist. Tatsächlich ist in Gofer ein Bezeichnungssystem für unendlich viele Typen vorhanden, alle diese Typen stehen damit zur Verfügung. Man kann diesen Sachverhalt so ausdrücken, daß in Gofer eine verallgemeinerte Rechenstruktur implementiert ist mit unendlich vielen Sorten und Wertemengen.

Das Bezeichnungssystem der Sorten basiert, genau wie bei den Funktionen, auf Konstruktoren, den sogenannten Typkonstruktoren, das sind

der Pfeil	`->`
die Tupelklammern	`( , )`
die Listenklammern	`[ ]`

Beipiel:

```
Int -> Bool
(Bool,Bool)
[ Char]
[ Int -> Int ] -> Bool
```

Typausdrücke der Form `A -> B -> C -> . . . -> Y -> Z` sind zulässig, und werden als rechtsgeklammert, d.h. als `A -> (B -> (C ->... -> (Y -> Z)...))` interpretiert. Die Typen `A, B,...` dürfen natürlich auch über Datenkeklarationen benutzerdefiniert sein. Eine Applikation einer Funktion `f` dieses Typs hat dann entsprechend die Form

```
f a b . . .
```

7. Funktionsdeklarationen

Wir betrachten nun die Deklarationen von Funktionen mittels Gleichungen genauer. Die allgemeine Form einer Gleichung sei an einem schematisierten Beipiel demonstriert Man beachte, daß in diesem Beispiel Formatierung benützt wird.

```
f  Muster     |   Bedingung1    =    Ausdruck1
              |   Bedingung2    =    Ausdruck2
               . . .
              |   Bedingung n   =    Ausdruck n
                      where     Deklarationen
```

Tatsächlich ist diese Form nur eine Kurzschreibweise für mehrere bedingte Gleichungen, je eine für jede Bedingung. Diese Schreibweise hat den Vorteil, daß man die linke Seite nicht wiederholen muß.

Die Bedingungen sind stets Boolsche Ausdrücke. Sie können samt senkrechter Strich entfallen. In diesem Fall ist die Gleichung unbedingt und entsprechend verkürzt. Für die letzte Bedingung kann auch `otherwise` geschrieben werden. Diese Bedingung ist immer dann erfüllt, wenn alle anderen Bedingungen nicht erfüllt sind. Die Ausdrücke auf den rechten Seiten der Gleichungen müssen alle den gleichen Typ haben und insbesondere mit dem Ergebnistyp der Funktion `f` übereinstimmen.

Die where-Klausel (`where` *Deklarationen*) kann ebenfalls entfallen. Die Deklarationen der where-Klausel können beliebige Funktions-, Typ- und Variablendeklarationen mit Musterbindung sein. Der Gültigkeitsbereich dieser Deklarationen umfaßt die Bedingungen und die Ausdrücke der rechten Seiten der Gleichungen.

Beipiel:

```
strichwert x  | x==['O']  = ""
              | x==['L']  = "|"
              | otherwise =
        strichwert( ix ) ++ strichwert( ix ) ++ strichwert( [last x] )
        where  ix = init(x)
```

8. Typdeklarationen

In einer Typdeklaration wird keinesfalls ein neuer Typ definiert, wie man vielleicht vermuten könnte. Eine Typdeklaration legt lediglich einen weiteren Namen (Synonym) für einen vorhandenen Typ in Form einer Typgleichung fest. Beispielweise kann ein Benutzer für die Werte des Typs `Int` die weitere Bezeichnung `Ganze_Zahl` festlegen. Dies geschieht dann durch die Deklaration

```
type Ganze_Zahl  = Int
```

Die Deklarationen `x::Int` und `y::Ganze_Zahl` ergeben in einer Gleichung `x = y` keinen Typkonflikt, weil die entsprechenden Wertemengen gleich sind.

Wir benutzen zunächst die folgende Form einer Typdeklaration.

`type` *Tpykonstruktor* `=` *Typkonstruktorterm*

Typkonstruktoren beginnen in der applikativen Schreibweise stets mit Großbuchstaben.

Beipiel:

```
type BoolMatrix  =  (Int,Int) -> Bool
```

Die obige Form der Typdeklaration ist in Gofer nicht die allgemeinste mögliche Form. Tatsächlich kann die Konstruktorbezeichnung mit Typparametern ergänzt werden und der Typkonstruktorterm kann diese Typvariablen enthalten. Diese Formen gehören zum Thema "polymorphe Typen" und sind nicht Gegenstand dieses Arbeitsblattes.

9. Datendeklarationen

Ganz im Gegensatz zur Typdeklaration werden in einer Datendeklaration tatsächlich neue Wertemengen definiert. Die Datendeklarationen gestatten es, Terme über neuen Funktionsbezeichnungen als neue Werte den vorhandenen Werten hinzuzufügen und für die Mengen dieser Werte neue Sortenbezeichnungen einzuführen.

Beipiel:

```
data Ganze_Zahl =  GanzeZahl  Int
data Farbe      =  Schwarz |  Rot  |  Gelb
data Baeume     =  Empty   |  Baum (Baeume, Baeume)
```

`GanzeZahl` ist ein Konstruktor, der eine ganze Zahl aus der Menge `Int` in einen neuen Wert transformiert. `GanzeZahl(1)` ist nicht identisch mit 1. Die Werte von `Schwarz, Rot, Gelb` sind neue Werte, die mit keinen anderen Werten identisch sind, auch nicht mit den Zeichenreihen `"Schwarz", "Rot", "Gelb"`. Diese neuen Werte haben einen Typ mit der Sortenbezeichnung `Farbe`. Die Werte vom Typ `Baeume` werden eineindeutig dargestellt von allen Termen, die sich mit den Konstruktoren `Baum` und `Empty` bilden lassen, z.B.

```
Baum ( Baum(Empty,Baum(Empty,Empty)) , Baum(Empty,Empty) )
```

Wir benutzen das folgende Schema für Datendeklarationen in formatierter Schreibweise.

$$\begin{array}{lll} \texttt{data} & \textit{Typkonstruktor} & = \textit{Konstruktor}\,11 \quad \textit{Typ}\,11 \quad \ldots \quad \textit{Typ}\,1n_1 \quad | \\ & & \qquad\qquad\quad \vdots \qquad\qquad\qquad\qquad\quad | \\ & & \textit{Konstruktor}\,m1 \quad \textit{Typ}\,m1 \quad \ldots \quad \textit{Typ}\,mn_m \end{array}$$

Die Typen an den Stellen *Typ ij* dürfen auch aus anderen Datendeklarationen stammen. Damit sind sogenannte "verschränkt rekursive" Datendeklarationen möglich.

Beipiel:

```
data WeissAmZug  =   Eroeffnung   |  SchwarzZieht  SchwarzAmZug
data SchwarzAmZug  =  WeissZieht  WeissAmZug
```

`WeissAmzug` bezeichnet die Menge aller Terme, auf die der Konstruktor `WeissZieht` anwendbar ist und umgekehrt. Der durch die Bezeichnungen nahegelegte Bezug zum Schachspiel soll hier nur als Gedächtnisstütze dienen. Es dürfte nicht schwerfallen, sich zu überlegen, wie die Mengen der Terme vom Typ `WeissAmZug` bzw. `SchwarzAmZug` aussehen.

Auch Datendeklarationen können in Gofer parametrisiert werden. Dazu kann der Typkonstruktor auf der linken Seite der Datengleichung mit einer Folge von Parameterbezeichnungen ergänzt werden, die dann auf der rechten Seiten an den Stellen *Typ ij* eingesetzt werden dürfen. Man bezeichnet die Datentypen mit Typparametern als "polymorphe" Datentypen. Polymorphe Datentypen sind nicht Gegenstand dieses Arbeitsblattes.

10. Termersetzung und Verzögerte Auswertung

Wir betrachten ein Goferprogramm mit den folgenden Deklarationen

```
first1 (x:xs) = x
generate x = x:( generate (x+1) )
```

und beobachten zunächst, wie Gofer den Aufruf `first1 (1:[1/0])` berechnet. Gofer kümmert sich keineswegs um den Teilausdruck `[1/0]`, sondern es versucht das äußerste Funktionssymbol, also `first1` mit Termersetzung aufzulösen. Dazu werden die Variablen des Musters `x:xs` auf der linken Seite der (ersten) Gleichung für `first1` an Teilausdrücke des Aufrufs gebunden mit dem eindeutigen Ergebnis

```
x = 1,    xs = [1/0]
```

Nun erfolgt die Termersetzung gemäß der Gleichung für `first1` mit dem Ergebnis 1. Damit endet aber schon die Berechnung. Der Teilterm `1/0` wird nicht ausgewertet. Bei einer Auswertung wäre auch eine Fehlermeldung die Folge. Betrachten wir nun die Funktion `generate` und den Aufruf `generate 1`. Die Variable `x` wird durch `x = 1` gebunden. Die erste Termersetzung führt zu `1:(generate (1+1))`. Als nächstes wird der Teilterm `generate (1+1)` durch Termersetzung ausgewertet mit dem Ergebnis

```
1: ( (1+1):(generate ((1+1)+1)  )
```

Dieser Auswertungsprozess wird offensichtlich nicht terminieren, es sei denn durch Abbruch. Ein Aufruf `generate 1` wird den Bildschirm mit einer nicht terminierenden Liste (unendlichen Liste) der natürlichen Zahlen füllen, solange bis der Benutzer den Prozess mit "ctrl C" abbricht, oder ein Speicherüberlauf stattfindet. Ganz anders aber reagiert Gofer auf die Eingabe des Ausdrucks

```
first1 (generate 1)
```

Das Ergebnis in diesem Fall ist das erste Listenelement von `generate 1`, nämlich 1. Die verzögerte Auswertung sorgt dafür, daß tatsächlich nur jene Terme ausgewertet werden, die für das Ergebnis notwendig sind und prüft nicht, ob der eingegebene Term insgesamt definiert ist nach der Semantik strikter Funktionen.

Die Auswertung einer Funktion `f` kann allerdings für ein Argument `x` erzwungen werden mit der einfachen Bedingung `x==x` in der ersten Gleichung für `f`.