Funktionale Programmierung mit Haskell

User Manual

Abstract

Die funktionale Programmierung ist durch eine sehr vereinfachte Programmierumgebung gekennzeichnet. Dadurch kann der Programmieranfänger sich auf wesentliche inhaltliche Konzepte der Programmierung konzentrieren. Die Programmiersprache *Haskell* befreit euch von den mitunter recht komplizierten Konzepten des Typsystems (z.B. gibt es in Haskell nur den Typ *num*, der die ganzen Zahlen (*Int*) und die Gleitkommazahlen (*Float*) beinhaltet) und der Ein/Ausgabe. Die Haskellsyntax ist sehr leicht zu erlernen, die Skripte sind selbsterklärend. Die Definition der Ausdrücke ist sehr stark an die Formulierung mathematischer Funktionen angelehnt. Durch die Einfachheit der Sprache kann der Programmieranfänger seine volle Konzentration auf das Erlernen wichtiger Datentypen und den entsprechenden Algorithmen auf ihnen lenken.

Haskell wurde Ende der 80er Jahre auf Grundlage der seit den 50er Jahren existierenden Sprache *Lisp* entwickelt. Angeregt durch Ideen von John Backus wurden auf Grundlage der sogenannten *lazy evaluation* (genaueres später) zahlreiche verschiedene funktionale Sprachen entwickelt. Eine weit verbreitete Sprache war *Miranda*, welche von David Turner entwickelt wurde. Da aber Turner sein Projekt nicht unter die GPL stellen wollte, konnte man *Miranda* nicht als Ausgangspunkt benutzen, um eine neue Sprache mit einheitlichen Standards zu entwickeln. Daher wurde *Haskell* als völlig neues System geschaffen. Der Name *Haskell* stammt von dem Mathematiker Haskell B. Curry, der mit dem sogenannten *Currying* wichtige theoretische Grundlagen der funktionalen Sprachen geschaffen hat.

Haskell umfasst ein vollständiges Programmiersystem mit Modulen für Grafik und Nebenläufigkeit. *Hugs* ist ein Haskellinterpreter und wird hier in der Version *Hugs98* verwendet. Ein anderer Interpreter ist der *GHCi.* Das ist der Glasgow Haskell Compiler. Den benutzen wir in der Schule.
Der Interpreter *GHCi* wird mit dem Befehl *ghci* in der Konsole oder über die Startumgebung gestartet. Nun können direkt Ausdrücke ausgewertet werden. (Z.B kann man *GHCi* als Taschenrechner benutzen.) Eine Übersicht über wichtige Systemkommandos erhält man mit :?. Um Algorithmen zu programmieren müssen diese in eine Datei mit der Endung *.hs (siehe unten) geschrieben werden.

Der große Vorteil von funktionalen Sprachen im allgemeinen und *Haskell* im speziellen ist die einfache Implementierung und Wartung großer Softwaresysteme. Der Quellcode ist kurz, klar und schwere Softwarefehler (sogenannte Seiteneffekte) werden auf ein Minimum reduziert.

Der folgende Text listet die wesentlichen Konzepte der funktionalen Sprachen auf. Es folgt eine Übersicht über einfache und zusammen-gesetzte Datentypen, sowie die wesentlichen Funktionen auf der Menge dieser Datentypen.

1. Wichtige Konzepte

1. In Haskell werden Definitionen in Form von Funktionen erzeugt. Diese Ausdrücke werden dann

vom Interpreter ausgewertet. Diese Funktionsdefinitionen (Anweisungen) können in einem Skript gesammelt werden und durch die Haskell-Laufzeitumgebung durch ghci *<Skriptname>.hs* ausgeführt werden.

2. Die am häufigsten verwendeten Strukturen sind rekursive Aufrufe. Rekursion bedeutet, dass sich eine Funktion selbst aufruft, bis ein Terminierungsfall eintritt (Nicht vergessen!!). Beispiele aus der Mathematik sind die Fakultäts- und die Fibonacchifunktion.

3. Jede Funktion besitzt eine Signatur (sie bekommt Parameter eines Typs und trägt selbst einen Typ), die auch bei der Programmierung mit angegeben werden sollte. Wichtige Eigenschaft von Funktionen ist der *Polymorphismus*.

4. Lokale Definitionen sind eine Möglichkeit Programmcode *top down* zu entwickeln und damit die Übersichtlichkeit zu erhöhen. Verwendet wird das Schlüsselwort *where*. Zu beachten ist die *Abseitsregel*.

5. Verwendung höherer Funktionen, welche wiederum aus einfachen Bibliotheksfunktionen aufgebaut sind.

2. Basisdatentypen

2.1 Bool

• Wertebereich von `Bool:  Bool::= False | True`

• Standardfunktionen: `||  :: Bool -> Bool -> Bool,` logisches Oder

• `&& :: Bool -> Bool -> Bool,` logisches Und

• `not :: Bool -> Bool`, logische Negation

Neben diesen wichtigen boolschen Operatoren gibt es Vergleichsoperatoren `(==, /=, >, <, >=, <=)` die aus der Mathematik bekannt sind. Genaueres zu Signatur und Syntax kann aus den entsprechenden Modulen (hier Prelude.hs) bzw. den Manuals entnommen werden.

2.2 Int und Integer

• Wertebereich umfasst alle ganzen Zahlen einfacher (4 Byte) und höherer (dynamische Speicherzuweisung, d.h. beliebige Genauigkeit)

• `+,-,* :: Int -> Int-> Int`, binäre Operationen, Hinweis: '/' liefert immer eine gebrochene Zahl, Bsp.: `6/2 = 3.0`
• `abs, negate, signum` tun das, was der Name vermuten lässt, `odd` bzw. `even` liefern je nach übergebener Zahl einen boolschen Wert
• `div, mod :: Int -> Int -> Int`, binäre Operationen für ganzzahlige Division und Rest

2.3 Float und Double

Gleitkommazahlen sind Zahlen der Form 3.14159, -23.17, 0.007, 55 oder 55.0. Neben dieser Notation darf auch die sog. wissenschaftliche Darstellung verwendet werden, bei der - wie in Programmiersprachen üblich - das e die Bedeutung „`x 10`$^{\text{hoch}}$“ hat.

3.14159e0 -2.317e1 7e-2 0.55e2

Auch hier einige wichtige vordefinierte Funktionen.

• `pi` (liefert die Zahl Pi) und `sqrt` (bekommen jeweils ein Float und berechnen die entsprechende Funktion)

2.4 Char

• Der Zeichentyp Char ist der geordnete Typ, dessen Wertebereich die Menge der ASCII-Zeichen mit den Codenummern 0..255 ist. Die Werte werden in einfachen Hochkommata eingeschlossen: 'A', '2'.Charakter sind auch diverse Steuerzeichen, z.B.

neue Zeile '\n'
Tabulator '\t'
Backslash '\\' (zu beachten ist die Maskierung mit den Backslash)

Zwei wichtige Funktionen sind *ord* und *chr. Ord* liefert zu einem ASCII-Zeichen den Dezimalwert, *chr* ist die inverse Funktion.

Fügt man mehrere Charakter aneinander, so erhält man einen String. Der formale Ausdruck für einen String ist [Char].
Für Zeichen und Zeichenketten stehen diese Operationen zur Verfügung:

Op.	Beispiel	Ergebnis	Bedeutung
`:`	`'H':"allo"`	`"Hallo"`	Anfügen eines Zeichens *vor* einem String
`++`	`"Ha" ++ "llo"`	`"Hallo"`	Aneinanderhängen zweier Strings
`head`	`head "Hallo"`	`'H'`	Erstes Zeichen (Kopf) eines Strings
`tail`	`tail "Hallo"`	`"allo"`	String ohne das erste Zeichen (Rumpf)

3. Benutzerdefinierte Datentypen

3.1 Listen

• Mit Buchstaben allein können noch keine Zeichenketten dargestellt werden. Dazu müssen mehrere Char aneinandergekoppelt (geconst) werden. Man erhält eine Liste von Char, also String = [Char]. (Diese Darstellung mit dem Gleichheitszeichen nennt man alias.)

• wichtigster Operator `(:) :: a -> [a] -> [a]`. Zu beachten ist der Polymorphismus. Dem cons-Operator ist es egal, welche Datentypen er zu einer Liste verbindet. Die Werte müssen nur gleich getypt sein.

• Bsp.: `'H': 'a':'l':'l':'o':[] = "Hallo"`

• wichtiges Konzept, welches häufig bei Listen Anwendung findet ist die Musteranpassung: Berechnung der Länge einer Liste

```
length [ ] = 0                    -- Terminierung
length (x:xs) = 1 + length xs     -- rekursiver Aufruf von length
```

Das Muster ist (x:xs). Haskell erkennt anhand dieses Musters das erste Element (den Kopf) und den Rest, wobei der Rest entsprechend der Signatur des cons-Operators eine Liste sein muss. Somit ist es möglich direkt auf die Teile einer Liste zuzugreifen, ohne Funktionen verenden zu müssen. Dies

geht in imperativen Sprachen allerdings bei statischen Arrays auch, allerdings ist der Nachteil der Längenbeschränktheit bei Listen in Haskell nicht vorhanden. Die Listen in Haskell sind praktisch dynamische Datenstrukturen, die einen einfachen Zugriff aufgrund der Existenz von Mustern gestatten.

• es gibt wichtige Bibliotheksfunktionen, die sich alle durch den vom System bereitgestellten cons-Operator darstellen lassen

• `++ :: [a] -> [a] -> [a],` ist die Verkettung zweier Listen (eine Art Listenaddition)

• `-- :: [a] -> [a] -> [a],` ist die Subtraktion zweier Listen,
Bsp.: `[1,2,3,4} -- [2,4] = [1,3]`

• `head:: [a]` → `a` liefert das erste Element einer nichtleeren Liste
• `last:: [a]` → `a` liefert das letzte Element einer nichtleeren Liste
• `tail:: [a]` → `[a]` liefert die Liste ohne das erste Element (Vor.: keine leeren Liste)
• `init:: [a]` → `[a]` liefert die Liste ohne das letzte Element (Vor.: keine leeren Liste)
• `reverse:: [a]` → `[a]` liefert die Liste in umgekehrter Reihenfolge
• weitere wichtige polymorphe Listenfunktionen (ausprobieren und Erklärung, sowie Beispiel selbständig notieren !!): `concat :: [[a]] -> [a] , drop, take :: Int -> [a] -> [a], max, min :: [a] -> a, replicate :: Int -> a -> [a], dropWhile, takeWhile :: (a -> Bool) -> [a] -> [a],`

Listen können neben der Erstellung mithilfe des cons-Operators (`[1,2,3] = 1:2:3:[]`) auch durch sogenannte *list comprehensions* erzeugt werden. Diese Listengeneratoren funktionieren wie Mengendefinitionen in der Mathematik. `[1,3,5,7,9,11,13,15,17] = [1,3 .. 17].` Neben der Kürze des Ausdruckes ist es auch möglich unendliche Listen zu verarbeiten, was durch das Konzept der *lazy evaluation* keine Probleme in der Verarbeitung macht.

3.2 Tupel

• Da Listen nur gleichartige Elemente enthalten dürfen, gibt es noch das Konzept der Tupel. Hier können verschiedene Datentypen enthalten sein.

(t_1..t_n) ist ein Typeltyp mit n Elementen, Bsp: `(-1, 'd', False)` ist vom Typ `(Int, Char, Bool)`

• `fst, snd` sind sogenannte Selektoren und liefern, entsprechend des Namens, die ersten bzw. zweiten Elemente des Tupels