Tag 6

Module, benannte Funktionen und Rekursion

Wir verlassen nun die abgeschlossene kleine Welt des interaktiven Interpreters und lernen, wie man richtige Haskell-Programme schreibt.
Im Interpreter haben wir bislang hauptsächlich Ausdrücke ausgewertet. Ein Haskell-Programm aber erlaubt wesentlich mehr als mit Ausdrücken allein möglich ist: Eigene Funktionen etwa (nicht nur mit Lambda-Abstraktion, sondern mit eigenem Namen) sind zwar im GHCi mittels der `let`-Konstruktion möglich, aber einfacher in einer Datei zu editieren. Zudem bietet ein Programm auch die Möglichkeit, eigene Datentypen und Prädikate (so wie `Num` und `Eq`) zu definieren, und Funktionen in wiederverwertbare Einheiten, sogenannte Module zu organisieren.
Ein Programm besteht aus mehreren Modulen. Jedes Modul steht in einer eigenen Datei. Die bereits vielzitierte `Prelude` ist selbst ein Modul, welches implizit in jedes andere Modul importiert wird und dessen Funktionen daher überall zur Verfügung stehen.
Haskell-Module können mit jedem beliebigen Editor erstellt werden, der Dateien im Textformat abspeichern kann. Wenn möglich, sollte in der Konfiguration die Verwendung von Tabulatoren ausgeschaltet oder die Tabulatorweite auf 8 Zeichen eingestellt werden. Haskell bietet nämlich im Vergleich zu vielen anderen Sprachen die Möglichkeit, das Layout des Programms dazu zu benutzen, Anweisungen zu gruppieren und dadurch unnötigen Ballast an Klammern und Trennsymbolen einzusparen. Das wird erst später wirklich relevant, aber eine Einstellung im Editor schon jetzt hilft, Mißverständnissen zuvorzukommen.
Der Dateiname sollte genau einen Punkt enthalten. Der Teil vor dem Punkt sollte dem Modulnamen entsprechen (und in Betriebssystemen mit Groß-/Kleinschreibungsunterscheidung mit einem Großbuchstaben beginnen, weil Haskell-Modulnamen mit einem Großbuchstaben beginnen müssen). Als Endung wählen wir `lhs`. Das steht für „literate Haskell“.
Jede Zeile in der Datei, die *nicht* mit `>` (also einem „Größer“-Zeichen gefolgt von einem Leerzeichen) beginnt, gilt als Kommentar. Zwischen Kommentar und Code-Zeilen muß immer noch mindestens eine Leerzeile liegen.
Hier im Text werde ich ebenfalls Haskell-Zeilen, die als Code für ein Module bestimmt sind, mit `>` einleiten, zur besseren Unterscheidung von Interaktionen im Haskell-Interpreter, die weiterhin mit dem vollen Interpreter-Prompt beginnen.
Ein Modul beginnt normalerweise mit einer Deklaration seines Namens, also etwa:

```
> module Tag6 where
```

Die Worte `module` und `where` sind sogenante „Schlüsselworte“ und müssen genau in dieser Form dort stehen. Der Modulname ist hier `Tag6`. Er muß, wie bereits erwähnt, mit einem Großbuchstaben beginnen.
Als vorläufig einzigen Inhalt von Modulen lernen wir Funktionsdefinitionen (mit zugehörigen Typsignaturen) kennen. Benannte Funktionen (und auch konstante Werte) können einfach mit Hilfe des Gleichheitszeichens eingeführt werden:

```
> identitaet = (\x -> x)
```

Wir müssen natürlich aufpassen, daß wir keine Namen verwenden, die schon von der Prelude in Beschlag genommen sind, etwa in diesem Fall der Name `id` für dieselbe Funktion.
Wenn wir das Modul jetzt unter `Tag6.lhs` abspeichern, dann können wir den GHCi oder den Hugs anschließend mit dem Dateinamen als Parameter aufrufen, und erhalten entweder eine Fehlermeldung über den Inhalt des Moduls, oder aber, anders als bisher, den Prompt `Tag6>` , der anzeigt, daß das Modul geladen ist und zur Verfügung steht. Alternativ kann der Interpreter mit der Anweisung `:l` auch „im Betrieb“ angewiesen werden, das Modul zu laden:

```
Prelude> :l Tag6

Tag6> identitaet 2
2
it :: Integer

Tag6> :t identitaet
forall t. t -> t
```

Tatsächlich können wir also den von uns eingeführten Namen `identitaet` nun verwenden, wie das Beispiel bereits demonstriert.

Aufgabe 1

Schreibe ein Modul mit Namen `Tag6Test`, speichere es in einer Datei `Tag6Test.lhs` ab und füge eine passende Modulkopfzeile sowie die Definition von `identitaet` ein. Lade dann das Modul auf die zwei beschriebenen Weisen in den Haskell-Interpreter.

Haskell bietet die Möglichkeit, selber definierten Funktionen eine Typsignatur zu verpassen. Dies ist, wie wir bereits gesehen haben, nicht nötig (es funktioniert ja auch so), aber empfehlenswert und guter Stil. Zum einen dient die Typsignatur bereits der Dokumentation der Funktion. Schon am Typ kann man einiges über das Verhalten der Funktion ablesen. Zum anderen sind die Typfehlermeldungen im allgemeinen präziser, wenn man Typsignaturen verwendet. Dann kann der Compiler nämlich konkret auf Diskrepanzen zwischen der Intention (der Typsignatur) und der tatsächlichen Realität (der Definition der Funktion) hinweisen, während er andernfalls erraten muß, welchen Typ die Funktion wohl haben sollte. Im Falle der Identitätsfunktion können wir schreiben:

```
> identitaet :: a -> a
```

Das `::` liest man als „hat den Typ“. Die Quantifizierung mittels `forall` wird weggelassen! Alle vorkommenden Variablen in einer Typsignatur gelten implizit als durch ein `forall` quantifiziert. Es ist sehr wahrscheinlich, daß in zukünftigen Sprachversionen von Haskell das `forall` explizit hingeschrieben wird oder es zumindest erlaubt ist, es explizit zu schreiben. Schon jetzt unterstützen sowohl Hugs als auch GHC diese Variante, wenn man einen „erweiterten Modus“ des Compilers aktiviert.
Normalerweise schreibt man die Typsignatur einer Funktion direkt vor deren Definition. Die Funktion zum Vertauschen von Paaren aus den Aufgaben der letzten Lektion etwa können wir in unserem Modul so definieren:

```
> swap :: (a,b) -> (b       ,a       )
> swap =  \pair -> (snd pair,fst pair)
```

```
Tag6> swap (1,'c')
('c',1)
it :: (Char, Integer)
```

Lambda-Abstraktion kann bei der Einführung eines Funktionsnamens auf angenehme Art und Weise abgekürzt werden. Wir können `swap` alternativ definieren als

```
> swap' :: (a,b) -> (b       ,a       )
> swap' pair     =  (snd pair,fst pair)
```

Auch mehrere Argumente (statt wiederholter Lambda-Abstraktion) können auf diese Art und Weise eingeführt werden, hier zum Beispiel eine Funktion, die drei Integer-Werte auf Gleichheit überprüft:

```
> triEqual :: Integer -> Integer -> Integer -> Bool
> triEqual a b c = a == b && b == c
```

Hier sehen wir auch gleich eine weitere Verwendung von Typsignaturen. Der deklarierte Typ kann *spezieller* sein als eigentlich nötig. *Ohne* Typsignatur würde der GHCi für `triEqual` den Typ `forall a. (Eq a) => a -> a -> a -> Bool` ableiten. (Normalerweise will man auch den allgemeinsten Typ haben, aber manchmal kann man auch durch solche Spezialisierungen die Verständlichkeit von Fehlermeldungen verbessern.)
Der wahre Vorteil, der durch die Einführung von Namen für unsere Funktionen entsteht, ist die Möglichkeit zur Verwendung von Rekursion. Rekursion ist in der funktionalen Programmierung allgegenwärtig und hat etwa den Stellenwert von Schleifen in imperativen Programmiersprachen. Als Beispiel schreiben wir die Funktion `enum`, die zwei Integer als Argumente nimmt, wobei der zweite größer sein sollte als der erste. Als Ergebnis wollen wir die Liste aller Zahlen, die zwischen der ersten und der letzten (einschließlich der Grenzen) liegen.

```
> enum :: Integer -> Integer -> [Integer]
> enum lower upper = if   lower == upper
>                    then [lower]
>                    else lower : enum (lower+1) upper
```

Zunächst überprüfen wir, ob die beiden Grenzen des Bereichs gleich sind. In diesem Fall können wir die richtige Antwort sofort geben: Die einelementige Liste, die diesen Begrenzungswert enthält. Andernfalls ist die „untere“ Schranke auf jeden Fall in der Ergebnisliste, und zwar als Kopfelement (wir verwenden daher den Cons-Operator `:`). Die Restliste ist genau der Bereich von `lower+1` bis `upper`, also können wir `enum` rekursiv dafür verwenden.

```
Tag6> enum 2 5
[2,3,4,5]
```

Eigentlich ist `lower` immer in der Ergebnisliste, unabhängig vom Ausgang des Vergleiches. Wir können deshalb alternativ schreiben:

```
> enum' :: Integer -> Integer -> [Integer]
> enum' lower upper = lower : if   lower == upper
>                             then []
>                             else enum (lower+1) upper
```

Diese Variante ist ein gutes Beispiel dafür, daß `if` in Haskell ein Teil von Ausdrücken und kein übergeordneter Befehl ist: Das gesamte `if`-Konstrukt kann hier problemlos als zweites Argument von ":" fungieren.
Natürlich funktionieren beide Funktionen nicht ordentlich, wenn die zweite Zahl kleiner ist als die erste. Die Eingabe

```
Tag6> enum 5 2
```

führt zu einer Endlosschleife, die man im Interpreter mit ⟨`Control-C`⟩ abbrechen kann.

Aufgabe 2

Schreibe `enum` so um (etwa als `enumX`), daß die beiden Grenzen nicht mit in der Ausgabeliste stehen. Auch hier muß die Funktion nicht funktionieren, wenn der zweite Wert nicht mindestens so groß ist wie der erste.

```
EnumX> enumX 1 1
[]

EnumX> enumX 2 3
[]

EnumX> enumX 3 6
[4,5]
```

Aufgabe 3

Schreibe eine Variante von `enumX`, die für den Fall, daß das zweite Argument kleiner ist als das erste, die Zahlen zwischen beiden Werten in umgekehrter Reihenfolge zurückliefert.

```
EnumX> enumX' 3 7
[4,5,6]

EnumX> enumX' 7 3
[6,5,4]
```

Die Prelude stellt die Vergleichsoperatoren <, <=, >= und > zur Verfügung.

Ein kleiner Tip zum einfacheren Entwickeln mit der interaktiven Umgebung: Wenn man ein Modul, welches von GHCi oder Hugs bearbeitet hat, gleichzeitig verändert, so kann man durch Eingabe von `:r` an der Eingabeaufforderung ein erneutes Laden des (veränderten) Moduls erreichen. Man muß also weder den Interpreter neu starten noch jedesmal das Modul explizit mit `:l` laden.

Aufgabe 4

Schreibe die Fakultätsfunktion `factorial :: Integer -> Integer` so, daß sie auf nichtnegativen Zahlen ein korrektes Ergebnis liefert:

```
Factorial> factorial 0
1
it :: Integer

Factorial> factorial 42
1405006117752879898543142606244511569936384000000000
it :: Integer
```

Lernziele Tag 6

- ☐ Haskell-Programme sind in Modulen organisiert.
- ☐ Module werden in Textdateien gespeichert und können von den Interpretern geladen werden.
- ☐ Module können Funktionsdefinitionen und Typsignaturen enthalten. Den Funktionen können nun Namen gegeben werden.
- ☐ Rekursion ist eines der wichtigsten Programmierprinzipien in der funktionalen Programmierung.