Kapitel 3
Funktionen und Operatoren

Bisher haben wir schon an der einen oder anderen Stelle Funktionen verwendet. Hier wollen wir dieses Konzept Schritt für Schritt erarbeiten und konkretisieren.

Um Funktionen in Haskell zu beschreiben, verwenden wir das Skriptformat `hs` (s. Abschn. 1.3.5), da sich die Kommentare größtenteils im Text des Buches und nicht im Programm befinden werden.

Es wäre sicherlich nicht hilfreich, alle in Haskell vordefinierten Funktionen in diesem Buch als lange Liste vorzustellen. Wir werden einige wichtige exemplarisch

M. Block, A. Neumann, *Haskell-Intensivkurs*,
DOI 10.1007/978-3-642-04718-3, © Springer 2011

erarbeiten und so ein Verständnis für die funktionale Programmierung entwickeln. Der Leser kann sich jederzeit in der Haskellspezifikation [40] eine Übersicht zu den vorhandenen Funktionen verschaffen und bei Bedarf die Funktionalität über die Suchmaschine Hoogle in Erfahrung bringen [55].

3.1 Funktionen definieren

Ein Funktionsname und alle Eingabeparameter beginnen in Haskell immer mit einem Kleinbuchstaben. Funktionen haben, wie wir bereits aus der Schule wissen, genau einen Rückgabewert.

Als erstes schauen wir uns Funktionen an, die Basisdatentypen als Eingabeparameter haben, also vom Typ `Bool`, `Int`, `Integer`, `Float` oder `Double` sind. Die Berechnungsvorschrift der `mystic`-Funktion soll an dieser Stelle nebensächlich sein:

```
mystic :: Int -> Int -> Int
mystic a b = div a 2 + (3*a - b)
```

Wir sehen zwei Programmzeilen. Schauen wir uns zunächst die Funktionsdefinition in der zweiten Zeile an. Dort sehen wir den Funktionsnamen `mystic` und anschließend die beiden Eingabeparameter `a` und `b`. Wenn die Funktion aufgerufen wird, z.B. mit `mystic 5 1`, dann werden `a` durch `5` und `b` durch `1` ersetzt und der rechte Teil der zweiten Zeile berechnet. Wir erhalten also als Ergebnis den Wert `16`, denn `div 5 2=2` und `2+(3*5-1)=16`.

In der ersten Zeile, die wir als Signatur bezeichnen, werden die Typen der Eingabeparameter und des Ausgabeparameters spezifiziert. Wird diese Zeile von links nach rechts gelesen, so bezeichnet der erste `Int` den Datentyp für den Platzhalter `a`, der zweite `Int` den Datentyp für den Platzhalter `b` und der dritte `Int` den Typ des Funktionsergebnisses.

Falls wir die allgemein gültige Signatur einer Funktion erfahren wollen, können wir diese in der Konsole mit dem Befehl `:type` oder kurz `:t` erfragen.

Nachdem wir das Skript geladen haben, können wir auf der Konsole die Signatur der Funktion `mystic` prüfen, die wir ja fest definiert haben:

```
Hugs> :t mystic
mystic :: Int -> Int -> Int
```

Wenn wir zwei Funktionen mit der gleichen Signatur haben, können wir diese auch zusammenfassend so notieren:

```
inkrement, dekrement :: Int -> Int
inkrement n = n+1
dekrement n = n-1
```

Beide Funktionen auf einen Zahlenwert angewendet, liefern wieder den Originalwert:

```
Hugs> inkrement (dekrement 5)
5
```

Die Zusammenfassung von Signaturen sollte aber nicht für größere Funktionen verwendet werden, da die Lesbarkeit darunter leiden kann.

3.1.1 Parameterübergabe

Eine Funktion muss keine Eingabeparameter haben, aber genau einen Rückgabewert. Die Funktion `zweiundvierzig` liefert immer den Zahlenwert `42`:

```
zweiundvierzig :: Int
zweiundvierzig = 42
```

Die verschiedenen Parameter sind immer positionstreu, dürfen also nicht vertauscht werden. Beispielsweise haben wir eine Funktion `intbool`, die einen `Int` und einen `Bool` als Eingabe erwartet:

```
intbool :: Int -> Bool -> Int
intbool a b = if (b==True) then a else 0
```

Die Funktion liefert den ersten Zahlenwert, wenn der zweite Wert `True` ist und ansonsten `0`. Von links nach rechts liest sich die Funktion wie folgt: Falls `b` den Wert `True` hat, liefere den Wert `a`, ansonsten den Wert 0.

Dann liefert eine korrekte Reihenfolge der Eingaben das erwartete Resultat:

```
Hugs> intbool 4 True
4
```

Sind beide Parameter allerdings beim Funktionsaufruf vertauscht, folgt eine Fehlermeldung von Hugs:

```
Hugs> intbool False 2
ERROR - Type error in application
*** Expression     : intbool False 2
*** Term           : False
*** Type           : Bool
*** Does not match : Int
```

Bei der Ausführung wurde die Typgleichheit überprüft und anstatt des erwarteten `Int` ein `Bool` gefunden.

3.1.2 Reservierte Schlüsselwörter

Bei der Namensgebung einer Funktion dürfen wir die folgenden Schlüsselwörter, deren Bedeutung wir im Laufe des Buches noch kennenlernen werden, nicht verwenden:

`case`, `class`, `data`, `default`, `deriving`, `do`, `else`, `if`, `import`, `in`, `infix`, `infixl`, `infixr`, `instance`, `let`, `module`, `newtype`, `of`, `then`, `type` und `where`.

Sie gehören zum Sprachumfang von Haskell und können auch nicht als Namen für die Parameter verwendet werden.

Das wäre also folglich falsch:

```
inkrementiere :: Int -> Int
inkrementiere do = do + 1
```

Wir erhalten folgende Fehlermeldung, wenn wir dieses Programm in Hugs laden wollen:

```
Hugs> :load "C:\\haskell_example_false.hs"
ERROR file:.\haskell_false.hs:2 - Syntax error in declaration
(unexpected token)
```

In jedem Fall sollten wir die Funktionsnamen so vergeben, dass die dahinter stehende Funktionalität erkennbar ist. Das fördert die Lesbarkeit erheblich.

3.1.3 Wildcards

Wenn einer der Parameter für die Berechnung der Funktion nicht benötigt wird, können wir das Symbol _ an dieser Stelle als Platzhalter (*wildcard*) schreiben.

Wenn wir die konstante Funktion `zweiundvierzig` noch einmal als Beispiel nehmen und einen Parameter vom Typ `Float` zusätzlich hinzufügen, können wir ihm die Bezeichnung `a` geben:

```
zweiundvierzig :: Float -> Int
zweiundvierzig a = 42
```

Da `a` aber keinen Einfluss auf die Ausgabe hat und schlichtweg uninteressant ist, können wir das durch die Verwendung einer Wildcard dem Leser deutlicher signalisieren:

```
zweiundvierzig _ = 42
```

In diesem Beispiel scheint die Verwendung eines zusätzlichen Parameters, der uns nicht interessiert, ein wenig sinnlos zu sein und das stimmt auch. Zu einem späteren Zeitpunkt, wenn wir damit beginnen größere Funktionen zu schreiben, werden wir aber oft von diesem Konzept Gebrauch machen und dann dessen Bedeutung besser verstehen.

3.1.4 Signaturen und Typsicherheit

Die Entwicklung von Funktionen in Haskell ist durch die sogenannte absolute Typsicherheit geprägt. Bei der Definition einer Funktion müssen wir eigentlich keine Signatur angeben. Haskell weiß automatisch, welche Signatur dem allgemein gültigen Fall der Eingaben entspricht.

Schauen wir uns dazu den folgenden Programmcode an:

```
inkrementiere :: Int -> Int
inkrementiere a = a + 1

dekrementiere a = a - 1
```

Wir haben die zwei Funktionen `inkrementiere` und `dekrementiere`, mit denen eine Zahl um `1` erhöht bzw. verkleinert wird. Die zweite Funktion ist ohne Signatur angegeben. Wenn wir dieses Programm in Hugs laden, funktioniert zunächst alles wie erwartet:

```
Hugs> inkrementiere 4
5

Hugs> dekrementiere 5
4

Hugs> :type inkrementiere
inkrementiere :: Int -> Int
```

Wenn wir allerdings die Signatur von `dekrementiere` erfahren wollen, erhalten wir die folgende Ausgabe:

```
Hugs> :type dekrementiere
dekrementiere :: Num a => a -> a
```

Hier steht `Num` für eine ganze Menge von Typen, eine sogenannte Typklasse. Zur Klasse `Num` gehören die bekannten Datentypen `Int`, `Integer`, `Float` und `Double`. Da für jeden dieser Datentypen die Addition definiert ist, erkennt Hugs, dass die Klasse `Num` den allgemeinen Fall darstellt. Was der Begriff Klasse ganz genau bedeutet, werden wir an einer späteren Stelle lernen.

Diese vier unterschiedlichen Signaturen kommen demnach in Frage:

```
dekrementiere :: Int     -> Int
dekrementiere :: Integer -> Integer
dekrementiere :: Float   -> Float
dekrementiere :: Double  -> Double
```

Damit ist die Funktion `dekrementiere` überladen (*overloaded*). Wir haben also den Fall, dass wir für den Eingabe- und den Ausgabeparameter den gleichen Typ verwenden wollen. Wir können das auch mit einem unbekannten Parameter `a` ausdrücken, wobei `a` der Klasse `Num` angehören muss:

```
dekrementiere :: Num a => a -> a
```

Wir wollen jetzt folgende Änderung an unserem Programm vornehmen und damit die beiden Methoden verknüpfen:

```
inkrementiere :: Int -> Int
inkrementiere a = a + 1

dekrementiere a = inkrementiere a - 2
```

Wenn wir Hugs jetzt wieder fragen, welche Signatur für `dekrementiere` gegeben ist, erhalten wir dieses Mal die folgende Antwort:

```
Hugs> :type dekrementiere
dekrementiere :: Int -> Int
```

Durch die Verwendung der Methode `inkrementiere`, die einen `Int` als Eingabe erwartet, schließt Hugs, dass auch in `dekrementiere` dieser Parameter ein `Int` sein muss.

3.1.5 Pattern matching

Der Übersicht halber kann eine Funktion auch in mehrere Definitionen zerlegt werden, das nennen wir Pattern matching. Dabei können Fallunterscheidungen der Eingaben bereits auf der linken Seite der Definition vorgenommen werden.

Angenommen, wir wollen die vier Fälle der Funktion `xor` (s. Abschn. 2.1.4) einzeln hinschreiben, dann könnten wir das wie folgt machen:

```
xor :: Bool -> Bool -> Bool
xor True  True  = False
xor True  False = True
xor False True  = True
xor False False = False
```

Dabei wird für eine Eingabe von oben beginnend geschaut, welche der Zeilen als erste passt. Sollten wir einen Fall abfragen, den wir in der Definition vergessen haben, dann liefert Hugs einen Fehler.

Angenommen, wir lassen die vierte Zeile der xor-Funktion weg und erfragen diese in Hugs, dann erhalten wir:

```
Hugs> xor False False
Program error: pattern match failure: xor False False
```

Wir können mit dem Pattern matching die Funktion xor jetzt auch wie folgt umstellen, um Zeilen zu sparen, denn Wildcards passen immer:

```
xor True  False = True
xor False True  = True
xor _     _     = False
```

Wenn eine der ersten beiden Zeilen passt, wird True zurückgegeben und in allen anderen Fällen False. Hier haben wir auch eine sinnvolle Anwendung der Wildcards.

3.1.6 *Pattern matching mit case*

Manchmal ist es hilfreich ein Pattern match nicht nur am Anfang einer Funktion machen zu können, sondern auch auf der rechten Seite. Für diesen Zweck gibt es case.

Hier ein Beispiel für die Funktion not:

```
not b = case b of
    True  -> False
    False -> True
```

Die Einrückung nach der case-Anweisung ist wichtig. Durch die konstante Einrückung erkennt Haskell, dass die folgenden Zeilen zu case gehören. Sie übernimmt die Funktion der geschwungenen Klammern in vielen anderen Programmiersprachen.

Auf der linken Seite von -> können die gewohnten Pattern verwendet werden. Im weiteren Verlauf des Buches werden wir case an einigen Stellen verwenden.

3.1.7 *Lokale Definitionen mit where*

Eine weitere Möglichkeit, die uns Haskell bietet, sind lokale Definitionen. Angenommen, wir schreiben eine Funktion, die eine Hilfsfunktion benötigt. Wir möchten diese Hilfsfunktion aber nicht nach außen sichtbar machen, sondern nur intern

verwenden und das genau in unserer Funktion. Dann können wir diese mit Hilfe des Schlüsselwortes `where` lokal definieren:

```
dekrementiere a = inkrementiere a -2
   where
      inkrementiere :: Int -> Int
      inkrementiere a = a + 1
```

Wird dieses Programm geladen, erkennt Hugs erfolgreich die nach außen sichtbare Funktion `dekrementiere`, aber `inkrementiere` bleibt verborgen:

```
Hugs> dekrementiere 3
2

Hugs> inkrementiere 2
ERROR - Undefined variable "inkrementiere"
```

Auch hier ist das konstante Einrücken wichtig. Dadurch erkennt Haskell, dass der nachfolgende Bereich zu `dekrementiere`, also zum Gültigkeitsbereich (*scope*) der Funktion gehört. An dieser Stelle sollte auch auf Tabulatoreinschübe verzichtet werden und stattdessen sollten Leerzeichen eingefügt werden. Der eingerückte Bereich muss die gleiche Anzahl an Leerzeichen aufweisen.

3.1.8 Lokale Definitionen mit let-in

Eine alternative Möglichkeit wird durch das `let-in`-Konstrukt angeboten. Um Formeln übersichtlicher und Hilfsfunktionen nach außen unsichtbar zu halten, können wir mit `let` lokale Funktionen definieren und diese in der darauf folgenden Zeile mit dem Schlüsselwort `in` verwenden.

Angenommen, es soll die folgende Funktion `formel` berechnet werden:

```
formel :: Int -> Int -> Int
formel a b = inkrementiere (2*a - 3*b) + dekrementiere (2*a - 3*b)
```

Es ist gleich zu sehen, dass eine Berechnung zweimal ausgeführt wird. Wir könnten diese durch ein `h` ersetzen und `h` lokal definieren:

```
formel a b = let h = (2*a - 3*b)
             in  inkrementiere h + dekrementiere h
```

Das erhöht neben der Lesbarkeit auch die Effizienz, denn die Berechnung für `h` findet jetzt nur einmal statt.

Auch unser Eingangsbeispiel können wir jetzt mit `let-in` angeben:

```
dekrementiere a = let inkrementiere a = a + 1
                  in  inkrementiere a - 2
```

3.1.9 *Fallunterscheidungen mit Guards*

Eine Möglichkeit der Fallunterscheidung haben wir über die Eingaben mittels Pattern matching kennengelernt. Es gibt auch die Möglichkeit, die Überprüfung innerhalb einer Funktionsdefinition vorzunehmen.

Wenn wir uns wieder die Funktion `xor` anschauen, dann wird die Überprüfung der Inhalte innerhalb der Funktion mit Hilfe der sogenannten Wächter (*guards*), symbolisiert durch `|`, vorgenommen:

```
xor x y
    | (x==True)  && (y==False) = True
    | (x==False) && (y==True)  = True
    | otherwise                = False
```

Auch hier werden die Fälle von oben nach unten betrachtet und der erste passende Fall genommen. Anders als beim Pattern matching haben wir hier aber die Möglichkeit, beliebig komplizierte Ausdrücke anzugeben. Mit `otherwise` haben wir einen Anker, der alle Fälle abfängt, die bis dorthin nicht gepasst haben. Damit haben wir wieder eine gültige Definition für `xor`.

Nach einem Guard steht eine Bedingung, also möglicherweise eine Formel, die entweder `True` oder `False` liefert. Wir können das Programm demnach noch etwas eleganter schreiben:

```
xor x y
    | x  && not y = True
    | not x && y  = True
    | otherwise   = False
```

Das Schlüsselwort `otherwise` verhält sich äquivalent zu `True`, denn sollten wir `True` als Bedingung einfügen, wird die entsprechende Zeile immer genommen. Testen wir das kurz auf der Konsole:

```
Hugs> otherwise
True
```

Für uns ist das Programm aber besser lesbar, wenn wir an dieser Stelle ein `otherwise` zu stehen haben.

Wir können uns auch ein eigenes Schlüsselwort `ansonsten` definieren

```
ansonsten :: Bool
ansonsten = True
```

und dieses statt `otherwise` verwenden:

```
xor x y
    | x  && not(y) = True
    | not(x) && y  = True
    | ansonsten    = False
```

3.1.10 Fallunterscheidungen mit if-then-else

In Bezug auf die Lesbarkeit hat sich bei den meisten imperativen Programmiersprachen das Konzept `if-then-else` durchgesetzt. Wenn (`if`) eine Bedingung erfüllt ist, dann (`then`) mache dies, ansonsten (`else`) mache das. Wir haben es also mit einer Fallunterscheidung zu tun, die sehr am natürlichen Sprachgebrauch angelehnt ist.

Ein Beispiel haben wir bereits kennengelernt:

```
intbool :: Int -> Bool -> Int
intbool a b = if (b==True) then a else 0
```

In der Bedingung wird überprüft, ob (`b==True`) wahr ist. Wenn das der Fall ist, wird `a` zurückgeliefert, ansonsten eine `0`. Da `b` selbst wieder vom Typ `Bool` ist, können wir die Bedingung auch noch kürzer notieren:

```
intbool a b = if b then a else 0
```

Hier die gleiche Funktion mit Guards:

```
intbool a b
   | b         = a
   | otherwise = 0
```

Die Fallunterscheidungen mit `if-then-else` sind in Haskell zwar möglich, erinnern aber stark an imperative Programmiersprachen und werden deshalb selten gebraucht. Wer die Schönheit der Guards zu schätzen gelernt hat, wird darauf auch nicht mehr verzichten wollen.

3.1.11 Kommentare angeben

In Abschn. 1.3.5 wurden die beiden Skriptformate `hs` und `lhs` vorgestellt. Für die Angabe von Kommentaren im `hs`-Format gibt es zwei Möglichkeiten. Ein einzeiliger Kommentar wird mit voranstehendem `--` gekennzeichnet:

```
-- Eine sehr kurze Formulierung der xor-Funktion:
xor :: Bool -> Bool -> Bool
xor x y = x /= y
```

Sollte ein größerer Kommentarbereich folgen oder beispielsweise eine Funktion auskommentiert werden, dann gibt es einen öffnenden `{-` und einen schließenden Kommentartag `-}`:

```
{- Funktion muss noch überprüft werden:
xor True  y = not y
xor False y = y
-}
```

3.2 Operatoren definieren

Die Verwendung von Operatoren erhöht die Lesbarkeit von Programmen. Wenn wir beispielsweise die Funktionen `plus` für Addition und `minus` für Subtraktion von ganzen Zahlen in Haskell anbieten, ist die Funktionsweise einer Formel teilweise schwer nachzuvollziehen.

```
plus, minus :: Int -> Int -> Int
plus  n m = n+m
minus n m = n-m
```

Der einfache Ausdruck $4-(1+1)+3$ sieht durch die entsprechende Klammerung wie folgt aus:

```
Hugs> plus (minus 4 (plus 1 1)) 3
5
```

Operatoren stellen typischerweise zweistellige Funktionen dar. In diesen Fällen können wir die Operatoren zwischen die Operanden schreiben (Infixschreibweise). In Haskell lassen sich Operatoren definieren und zweistellige Funktionen sogar als Operatoren verwenden.

3.2.1 Assoziativität und Bindungsstärke

Einige der in Tabelle 3.1 vorhandenen Operatoren haben wir bereits kennengelernt.

Tabelle 3.1 Die Assoziativitäten der Operatoren sind wichtig, um die Abarbeitungsreihenfolge zu kennen [18]

Operator	Assoziativität
`!!`	links
`.`	rechts
`^`	rechts
`**`	rechts
`*`	links
`/`	links
`'div'`	links

Operator	Assoziativität
`'mod'`	links
`+`	links
`-`	links
`:`	rechts
`++`	rechts
`&&`	rechts
`||`	rechts

Durch die angegebene Assoziativität wissen wir nun, in welcher Reihenfolge die Operatoren abgearbeitet werden und auf welche Klammern wir verzichten können.

Neben der Assoziativität ist aber auch die Bindungsstärke (s. Tabelle 3.2) für eine bessere Lesbarkeit wichtig.

Tabelle 3.2 Benötigte und überflüssige Klammern lassen sich leicht durch die Bindungsstärke ermitteln [18]

Operator	Bindung
`!!`	9
`.`	9
`^`	8
`**`	8
`*`	7
`/`	7
`'div'`	7
`'mod'`	7

Operator	Bindung
`+`	6
`-`	6
`:`	5
`++`	5
`/=`	4
`==`	4
`<`	4
`<=`	4

Operator	Bindung
`>`	4
`>=`	4
`'elem'`	4
`'notElem'`	4
`&&`	3
`||`	3

Alle Funktionen haben dabei die Bindungsstärke 10. Schauen wir uns die folgenden Aufrufe im Vergleich an:

```
Hugs> (((4^3)+5)*(2^8))
17664

Hugs> (4^3+5)*2^8
17664
```

In beiden Fällen wird die Formel $(4^3 + 5) \cdot 2^8$ berechnet, aber aufgrund der Bindungsstärke im zweiten Beispiel auf unnötige Klammern verzichtet.

3.2.2 Präfixschreibweise – Operatoren zu Funktionen

Operatoren werden automatisch in der Infixschreibweise notiert, das Symbol wird demnach zwischen die Operanden geschrieben. Die Operation `&&` mit zwei Operanden notieren wir beispielsweise mit `x&&y`.

Ein Operator lässt sich aber auch als Funktion schreiben und bei Funktionen steht der Name immer vorn. Wir bezeichnen das dann als Präfixschreibweise.

Wollen wir also den Ausdruck `x&&y` in Präfixschreibweise notieren, so müssen wir eine Klammer um den Operator schreiben: `(&&) x y`. Beide Schreibweisen sind äquivalent zu verwenden. Der Vorteil der Präfixschreibweise bei Operatoren ergibt sich bei der Signaturdefinition.

So wird beispielsweise der `&&`-Operator definiert mit:

```
(&&) :: Bool -> Bool -> Bool
(&&) True True = True
(&&) _    _    = False
```

Bei der Signaturdefinition verwenden wir die Präfixschreibweise, eine andere Möglichkeit gibt es nicht.

Schauen wir uns noch ein kleines Anwendungsbeispiel an:

```
Hugs> True && True == (&&) True True
True
```

3.2.3 *Infixschreibweise – Funktionen zu Operatoren*

Funktionen mit zwei Eingabeparametern, die immer in Präfixschreibweise notiert werden, lassen sich auch als Operatoren umfunktionieren. Dazu wird der Funktionsname `func` in `func x y` mit accent grave versehen und zwischen die Operanden geschrieben `x 'func' y`.

Schauen wir uns dafür ein Beispiel zu dem `xor`-Operator aus Abschn. 2.1.4 an:

```
Hugs> xor True False
True

Hugs> True 'xor' False
True
```

Eine weitere Umwandlung zurück zur Präfixschreibweise ist nicht möglich und führt zu einer Fehlermeldung:

```
Hugs> ('xor') True False
ERROR - Syntax error in expression ...
```

3.2.4 *Eigene Operatoren definieren*

In Abschn. 3.2.2 haben wir schon gesehen, wie einfach es ist, einen Operator zu definieren. Die Wahl eines Symbols oder einer Kombination aus Symbolen darf mit den bereits definierten nicht kollidieren.

Als einfaches Beispiel wollen wir einen neuen Operator für die `xor`-Funktion definieren. Dazu könnten wir das Symbol # verwenden:

```
(#) :: Bool -> Bool -> Bool
(#) x y = (x || y) && (not (x && y))
```

Wir können den neu definierten Operator analog zu den bereits bekannten einsetzen:

```
Hugs> True # False
True
```

Bei der Operator-Definition handelt es sich in der Regel um zweistellige Funktionen. Es ist prinzipiell auch möglich, Operatoren mit mehr Operanden zu definieren. Um diese Definitionen zu verstehen, sind allerdings Konzepte nötig, die wir an dieser Stelle noch nicht behandelt haben.

Auch für selbstdefinierte Operatoren ist es möglich, die Assoziativität sowie die Bindungsstärke festzusetzen. Hierfür dienen die Schlüsselwörter `infixl` und `infixr` für Links- bzw. Rechtsassoziativität.

Wir können so beispielsweise die Punkt-vor-Strichrechnung aufheben, indem wir eigene Operatoren für Addition und Multiplikation definieren:

```
infixr 8 .+.
infixr 7 .*.

(.+.) a b = a + b
(.*.) a b = a * b
```

Mit diesen Definitionen können wir jetzt etwas überraschende Ergebnisse bei der Termauswertungen beobachten:

```
Hugs> 3 + 5 * 7
38

Hugs> 3 .+. 5 .*. 7
56
```

Durch die neue Bindungsstärke haben wir jetzt statt der üblichen Klammerung $3 + (5 \cdot 7)$ den Term $(3 + 5) \cdot 7$ ausgewertet.

3.3 Lambda-Notation

Eine interessante Möglichkeit, namenlose Funktionen zu definieren, bietet die Lambda-Notation an. Sinnvoll ist der Einsatz dann, wenn Funktionsargumente nur in einem bestimmten Kontext verwendet werden. Eine separate Definition ist in solchen Fällen nicht notwendig.

Wenn wir beispielsweise eine Summenfunktion mit n Argumenten nehmen

```
f x1 x2 ... xn = x1+x2+...+xn
```

können wir durch die Lambda-Notation auf den Namen `f` verzichten und dieselbe Funktion wie folgt angeben:

```
\x1 x2 ... xn -> x1+x2+...+xn
```

Beide haben die gleiche Bedeutung. Da das Symbol λ (Lambda) in einem normalen Editor nicht so leicht verwendet werden kann, wird dieses durch das Symbol ‚\' repräsentiert.

Namenlose Funktionen dürfen wir nicht einfach in unserem Haskellskript angeben. Das würde auch nicht sinnvoll sein, da wir später nicht mehr auf sie zugreifen könnten.

Sollten wir jetzt beispielsweise das Quadrat als namenlose Funktion einbauen, könnte das so aussehen:

```
quadrat = \x -> x*x
```

Den Unterschied können wird auf der Konsole nicht sehen:

```
Hugs> quadrat 4
16
```

Später, wenn wir Funktionen als Argumente verwenden, werden wir die Einsatzmöglichkeiten der Lambda-Notation schätzen lernen.

3.4 Übungsaufgaben

Aufgabe 1) Prüfen Sie, ob die folgende Definition der Funktion `xor` korrekt ist:

```
xor True  y = not y
xor False y = y
```

Aufgabe 2) Schreiben Sie mit `if-else` eine Funktion `greater a b`, die `a` zurückliefert, wenn `a>=b` ist und `b` sonst.

Aufgabe 3) Erstellen Sie ein Skript mit allen Funktionen, die in diesem Kapitel vorgestellt wurden und überprüfen Sie die Funktionalität.

Aufgabe 4) Definieren Sie einen Operator `opNAND`, der die folgende Wertetabelle erfüllt:

a	b	a opNAND b
0	0	1
0	1	1
1	0	1
1	1	0

Warum besitzen NAND und NOR in der Informatik eine so große Bedeutung? Recherchieren Sie.