4 Nachtrag: Haskell

4.1 Boolesche Logik

Einfache boolesche Operatoren

Anhand der Booleschen Logik und von Operationen auf Bit-Folgen werde weitere Beispiele für die Funktionalität von Haskell betrachtet.

Nachfolgend sind einige einfachen, d.h. unäre und binäre, boolesche Operatoren und deren Wahrheitswertetabellen angegeben.

Hinweis. 0 repräsentiert den Wahrheitswert *false*, 1 den Wahrheitswert *true*.

NOT

A	X
0	1
1	0

AND

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

NAND

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

XOR

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

OR

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

NOR

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

In **`prelude`** sind der Typ `Bool` und zu **`NOT`**, **`AND`** und **`OR`** passende Funktionen und Operatoren definiert.

```
ghci> :t not
not :: Bool -> Bool
ghci> :t (&&)
(&&) :: Bool -> Bool -> Bool
ghci> :t (||)
(||) :: Bool -> Bool -> Bool
```

Mehrstellige boolesche Funktionen

Es lassen sich, analog zu den einfachen booleschen Funktionen, auch mehrstellige boolesche Funktionen, d.h. Funktionen in mehreren Variablen, definieren.

$$AND\ (A_1, \ldots, A_n) = \begin{cases} 1 & \text{wenn alle } A_1, \ldots A_n \text{ gleich 1 sind} \\ 0 & \text{sonst} \end{cases}$$

$$OR\ (A_1, \ldots, A_n) = \begin{cases} 0 & \text{wenn alle } A_1, \ldots A_n \text{ gleich 0 sind} \\ 1 & \text{sonst} \end{cases}$$

$$NAND\ (A_1, \ldots, A_n) = \begin{cases} 0 & \text{wenn alle } A_1, \ldots A_n \text{ gleich 1 sind} \\ 1 & \text{sonst} \end{cases}$$

$$NOR\ (A_1, \ldots, A_n) = \begin{cases} 1 & \text{wenn alle } A_1, \ldots A_n \text{ gleich 0 sind} \\ 0 & \text{sonst} \end{cases}$$

Ebenfalls in **`prelude`** definiert sind Funktionen, die zu **`AND`** und **`OR`** in mehreren Variablen passen. Wobei die Haskell-Funktionen **`and`** und **`or`** auf Listen von Booleschen Werten angewendet werden.

Beispiele

```
ghci> and [True, False, False]
False
ghci> and [True, True, True]
True
ghci> or [True, False, False]
True
ghci> or [False, False False]
False
```

4.2 Funktionen auf Listen und Tupeln

Zum Testen von Funktionen und Operatoren ist es nützlich eine Testfunktion zu definieren.

Die Testfunktion soll als Funktionswert eine Zeichenkette (**String**) zurückliefert, die den Funktionswert der zu testenden Funktion, bei Anwendung auf konkrete Argumente, repräsentiert.

Den erste Schritt dazu realisiert die Funktion **show**, die für fast alle Typen definiert ist und als Funktionswert die Zeichenkettenrepräsentation des Arguments als **String** zurückliefert.

```
ghci> show( True )
"True"
ghci> show( and[False, False, False] )
"False"
```

Mehrere Strings kann man, wie Listen, mit dem Operator **(++)** verbinden (*concat*).

Der Zeilenumbruch wird über `"\n"` kodiert.

```
ghci> show( and[False, False, False] ) ++ "\n" ++
      show( and[True, True, True] )
"False\nTrue"
```

Eine formatierte Ausgabe erhält man mit der Funktion **putStrLn**.

```
ghci> putStrLn( show( and[False, False, False] ) ++ "\n" ++
      show( and[True, True, True] ) )
False
True
```

Beispiel

```
-- test_bool.hs
testAND :: String
testAND =
  show( and[False, False, False] ) ++ "\n" ++
  show( and[False, False, True ] ) ++ "\n" ++
  show( and[False, True , False] ) ++ "\n" ++
  show( and[False, True , True ] ) ++ "\n" ++
  show( and[True , False, False] ) ++ "\n" ++
  show( and[True , False, True ] ) ++ "\n" ++
  show( and[True , True , False] ) ++ "\n" ++
  show( and[True , True , True ] )
```

```
ghci> :l test_bool.hs
...
ghci> putStrLn testAND
False
False
False
False
False
False
False
True
```

Soll eine Wahrheitswertetabelle ausgegeben werden, geht das mit Haskell auch kompakter und insbesondere ohne Code-Vervielfachung.

Zuerst wird für eine beliebige Funktion (`[Bool] -> Bool`), die eine Liste von Wahrheitswerten auf genau einen Wahrheitswert abbildet, und eine Liste von Wahrheitswerten (`[Bool]`) eine Tabellenzeile (`String`) erzeugt.

```
table_row :: ([Bool] -> Bool) -> [Bool] -> String
table_row f xs = show xs ++ " : " ++ show (f xs)
```

```
ghci> table_row and [True, False, True]
"[True,False,True] : False"
```

Die ganze Tabelle wird erzeugt, indem eine Liste von Listen von Wahrheitswerten (`[[Bool]]`) rekursiv durchlaufen wird.

In jedem Schritt wird von `head` aus der umschließenden Liste die erste Liste (`[Bool]`) geliefert, diese wird mit `table_row` ausgegeben.

Die umschließenden Liste ohne die erste Liste wird von **tail** geliefert. Mit dieser kleineren Liste wir die Rekursion durchgeführt.

Der Basisfall, für den keine Rekursion mehr nötig ist, tritt ein, wenn die umschließende Liste leer ist (wird getestet mit **null**).

```
tableA :: ([Bool] -> Bool) -> [[Bool]] -> String
tableA f xs
  | null xs    = ""
  | otherwise = table_row f (head xs) ++ "\n" ++
                tableA f (tail xs)
```

Die Funktionen **head**, **tail** und **null** sind nicht die beste Option Listen zu bearbeiten. Vorzuziehen ist eine Lösung mit *pattern matching*.

```
table :: ([Bool] -> Bool) -> [[Bool]] -> String
table f [] = ""
table f (x:xs) = table_row f x ++ "\n" ++ table f xs
```

Das *pattern* **[]** beschreibt die leere Liste.

Das *pattern* **(x:xs)** passt **nur** zu einer **nicht leeren** Liste. Im folgende können **x**, das erste Element der Liste, und **xs**, die - möglicherweise leere - Liste ohne das erste Element, einzeln benutzt werden.

Ähnlich wie mit **where** lassen sich mit **let** Platzhalter für Ausdrücke definieren.

Haskell bietet die Möglichkeit Listen auf sehr viele verschiedene Arten zu definieren, eine wird im Folgenden verwendet.

Genau erläutert wird diese sogenannten *list comprehensions* später.

```
ghci> let bool_tri =  [[a,b,c] | a <- [False, True],
                                 b <- [False, True],
                                 c <- [False, True]]
ghci> putStrLn ( table and bool_tri )
[False,False,False] : False
[False,False,True] : False
[False,True,False] : False
[False,True,True] : False
[True,False,False] : False
[True,False,True] : False
[True,True,False] : False
[True,True,True] : True
```

Haskell stellt (in jedem Modul) eine vordeklarierte Funktion **main** bereit, unter der sich, z.B. in einem **do**-Block, eine Folge von Ausdrücken zusammenfassen lässt.

Die Ausdrücke in der Funktion **main** erzeugen Ausgaben (z.B. **putStrLn**), erwarten Eingaben oder haben keinen Wert (z.B. **let**).

```
main = do
  let bool_tri =  [[a,b,c] | a <- [False, True],
                             b <- [False, True],
                             c <- [False, True]]
  putStrLn ( table and bool_tri )
```

```
ghci> main
[False,False,False] : False
[False,False,True] : False
[False,True,False] : False
[False,True,True] : False
[True,False,False] : False
[True,False,True] : False
[True,True,False] : False
[True,True,True] : True
```

Folgen von Bits

Ein Wahrheitswert (*false*/*true*) kann auch als **Bit** (0/1) interpretiert werden.

Mit Hilfe der Booleschen Logik sind dann auch Operationen auf Folgen von Bits (z.B. 8 Bit = 1 **Byte**, 4 Bit = 1 **Nibble**) möglich.

Der Test auf Gleichheit für zwei gleich lange Folgen von Bits kann wie folgt realisiert werden.

$$\begin{aligned}&equals(x_{n-1},\ldots,x_0,y_{n-1},\ldots y_0)\\&= NOR\left((x_{n-1}\ XOR\ y_{n-1}),\ldots,(x_0\ XOR\ y_0)\right)\end{aligned}$$

Nicht definiert in **prelude** ist ein Funktion oder ein Operator für **XOR**.

Man kann selbst einen passenden Operator für **XOR** definieren.

```
(<+>) :: Bool -> Bool -> Bool
(<+>) a b = (a || b) && (not (a && b))
```

Zum Testen des Operators wird eine Testfunktion (**table**), mit Hilfsfunktion (**table_row**), definiert.

```
table_rowA :: (Bool -> Bool -> Bool) -> (Bool, Bool) -> String
table_rowA f xt = show(xt) ++ " : " ++ show(f (fst xt) (snd xt))
```

Hinweis

- Das erste Element eines Paars (2-Tupels) wird von **fst**, das zweite von **snd** geliefert.

Das lässt sich auch mit *pattern matching* umsetzen.

```
table_row :: (Bool -> Bool -> Bool) -> (Bool, Bool) -> String
table_row f (x,y) = show (x,y) ++ " : " ++ show (f x y)
```

Hinweis

- Das *pattern* **(x,y)** steht für ein Paar, die Elemente **x** und **y** können nachfolgend auch einzeln verwendet werden.

Testfunktion mit **main**.

```
table :: (Bool -> Bool -> Bool) -> [(Bool, Bool)] -> String
table f []     = ""
table f (x:xs) = table_row f x ++ "\n" ++ table f xs

-- test XOR operator
main = do
  let bool_duo =  [(a,b) | a <- [False, True],
                           b <- [False, True]]
  putStrLn ( table (<+>) bool_duo )
```

Der Test ist erfolgreich.

```
ghci> main
(False,False) : False
(False,True) : True
(True,False) : True
(True,True) : False
```

Für den Test auf Gleichheit von zwei Bit-Folgen wird noch ein mehrstelliges NOR benötigt.

Man kann die aus der Mathematik bekannte **Komposition** (Hintereinanderausführung) von Funktionen verwenden.

Seien X, Y, Z beliebige Mengen und $f : X \to Y$ und $g : Y \to Z$ Funktionen, dann ist die Funktion $g \circ f : X \to Z$ die Komposition von f und g und es gilt folgendes.

$$\Big(g \circ f\Big)(x) = g\Big(f(x)\Big)$$

In Haskell wird die Komposition mit dem Punkt-Operator `(.)` realisiert.

```
nor :: [Bool] -> Bool
nor = not.or
```

Für eine kompaktere Darstellung wird mit Hilfe des Statements **`type`** ein Synonym **`Nibble`**, für ein 4-Tupel von Wahrheitswerten, eingeführt.

```
type Nibble = (Bool, Bool, Bool, Bool)
```

Jetzt kann eine Funktion zum Test auf Gleichheit von zwei **`Nibble`** definiert werden.

```
equals :: Nibble -> Nibble -> Bool
equals (a3, a2, a1, a0) (b3, b2, b1, b0)
  = nor [a3 <+> b3, a2 <+> b2, a1 <+> b1, a0 <+> b0]
```

Die Funktion kann wie üblich getestet werden.

```
ghci> equals (True, False, False, False)
             (True, False, False, False)
True
ghci> equals (True, False, False, False)
             (False, False, False, False)
False
```