

INFORMATIK I

Tutorium 7 — 29. November 2024

Von Listen und Screenshots

Name Last
Universität Münster



L^AT_EX-Vorlage von
Florian Sihler

Übungsblatt 6

1 ○

MENGEN IN HASKELL

Wir möchten Mengen in Haskell verwalten.

MENGEN IN HASKELL

Wir möchten Mengen in Haskell verwalten.

→ **Keine Reihenfolge und keine Duplikate!**

MENGEN IN HASKELL

Wir möchten Mengen in Haskell verwalten.

→ **Keine Reihenfolge und keine Duplikate!**

Aufgabe 1: a)

`is_empty`: Prüft, ob eine Menge leer ist.

MENGEN IN HASKELL

Wir möchten Mengen in Haskell verwalten.

→ **Keine Reihenfolge und keine Duplikate!**

Aufgabe 1: a)

`is_empty`: Prüft, ob eine Menge leer ist.

Lösung 1: a)

```
is_empty :: [Integer] -> Bool
is_empty [] = True
is_empty _ = False
```

Aufgabe 1: b)

`is_elem`: Prüft, ob ein Wert in einer Menge enthalten ist.

MENGEN IN HASKELL

Aufgabe 1: b)

`is_elem`: Prüft, ob ein Wert in einer Menge enthalten ist.

Lösung 1: b)

```
is_elem :: Integer -> [Integer] -> Bool
is_elem _ [] = False
is_elem x (y:ys)
    | x == y = True
    | otherwise = is_elem x ys
```


Aufgabe 1: c)

`is_subset`: Prüft, ob die erste Menge in der zweiten enthalten ist.

MENGEN IN HASKELL

Aufgabe 1: c)

`is_subset`: Prüft, ob die erste Menge in der zweiten enthalten ist.

Lösung 1: c)

```
is_subset :: [Integer] -> [Integer] -> Bool
is_subset [] _ = True
is_subset _ [] = False
is_subset (x:xs) set2
    | is_elem x set2 = is_subset xs set2
    | otherwise = False
```

Aufgabe 1: d)

`is_equal`: Prüft, ob zwei Mengen gleich sind.

MENGEN IN HASKELL

Aufgabe 1: d)

`is_equal`: Prüft, ob zwei Mengen gleich sind.

Lösung 1: d)

```
is_equal :: [Integer] -> [Integer] -> Bool
is_equal set1 set2 = is_subset set1 set2 && is_subset set2 set1
```

Aufgabe 1: e)

`delete`: Löscht ein Element aus einer Menge.

Aufgabe 1: e)

`delete`: Löscht ein Element aus einer Menge.

Lösung 1: e)

```
delete :: Integer -> [Integer] -> [Integer]
delete _ [] = []
delete x (y:ys)
    | x == y = ys
    | otherwise = y : delete x ys
```

Aufgabe 1: f)

`insert`: Fügt ein Element zu einer Menge hinzu.

Aufgabe 1: f)

`insert`: Fügt ein Element zu einer Menge hinzu.

Lösung 1: f)

```
insert :: Integer -> [Integer] -> [Integer]
insert x [] = [x]
insert x set
    | is_elem x set = set
    | otherwise = x : set
```


Aufgabe 1: g)

`union`: Bildet die Vereinigung zweier Mengen.

MENGEN IN HASKELL

Aufgabe 1: g)

`union`: Bildet die Vereinigung zweier Mengen.

Lösung 1: g)

```
union :: [Integer] -> [Integer] -> [Integer]
union [] set2 = set2
union set1 [] = set1 -- effizienzhalber
union (x:xs) set2 = insert x (union xs set2)
```

Aufgabe 2: a)

Programmieren Sie eine Funktion, die für einen Integer-Wert prüft, ob dieser eine Primzahl ist.

WARUM SO FAUL?

Aufgabe 2: a)

Programmieren Sie eine Funktion, die für einen Integer-Wert prüft, ob dieser eine Primzahl ist.

Lösung 2: a)

```
is_prime :: Integer -> Bool
is_prime n
  | n <= 1 = False
  | n == 2 = True
  | n `mod` 2 == 0 = False
  | otherwise = null [ x | x <- [3,5 .. floor (sqrt (↵
    fromIntegral n))] , n `mod` x == 0 ]
```

WARUM SO FAUL?

Aufgabe 2: a)

Programmieren Sie eine Funktion, die für einen Integer-Wert prüft, ob dieser eine Primzahl ist.

Lösung 2: a)

```
is_prime :: Integer -> Bool
```

```
is_prime n
```

```
  | n <= 1 = False
```

```
  | n == 2 = True
```

```
  | n `mod` 2 == 0 = False
```

```
  | otherwise = null [ x | x <- [3,5 .. floor (sqrt (↵  
    fromIntegral n))] , n `mod` x == 0 ]
```

$$\forall n = x \cdot y : \\ x \geq \sqrt{n} \implies y \leq \sqrt{n}$$

Aufgabe 2: b)

Definieren Sie unter Nutzung von `is_prime` eine Liste aller Primzahlen.

WARUM SO FAUL?

Aufgabe 2: b)

Definieren Sie unter Nutzung von `is_prime` eine Liste aller Primzahlen.

Lösung 2: b)

```
primes :: [Integer]
primes = [ n | n <- [2..], is_prime n ]
```

WARUM SO FAUL?

Aufgabe 2: b)

Definieren Sie unter Nutzung von `is_prime` eine Liste aller Primzahlen.

Lösung 2: b)

```
primes :: [Integer]
primes = [ n | n <- [2..], is_prime n ]
```

oder effizienter:

```
primes :: [Integer]
primes = [2] ++ [ n | n <- [3,5 ..], is_prime n ]
```


Aufgabe 2: c)

Extrahieren Sie die 42. Primzahl.

WARUM SO FAUL?

Aufgabe 2: c)

Extrahieren Sie die 42. Primzahl.

Lösung 2: c)

```
prime_no :: Int -> Integer  
prime_no n = last (take n primes)
```

WARUM SO FAUL?

Aufgabe 2: c)

Extrahieren Sie die 42. Primzahl.

Lösung 2: c)

```
prime_no :: Int -> Integer  
prime_no n = last (take n primes)
```

```
> prime_no 42  
181
```

REALLY? EIN SCREENSHOT?

Aufgabe 3

Ihr bekommt 5 Punkte, um Java und Eclipse zu installieren. 😐

REALLY? EIN SCREENSHOT?

Aufgabe 3

Ihr bekommt 5 Punkte, um Java und Eclipse zu installieren. 😐

Lösung 3

Aufgabe 4

Nutzen Sie in dieser Aufgabe die Kommandozeile zum Kompilieren und Ausführen von Java-Code. Lesen Sie zwei Zahlen von der Kommandozeile ein und speichern Sie diese als Variablen. Tauschen Sie die Werte der beiden Variablen und geben Sie den größeren Wert aus.

Lösung 4: a) + b) + c)

Lösung 4: $a) + b) + c)$

```
// Deklaration (a)  
int x, y;
```


Lösung 4: a) + b) + c)

```
// Deklaration (a)
int x, y;

// Eingabe (b)
x = IOTools.readInt("Eingabe_x:_");
y = IOTools.readInt("Eingabe_y:_");
```

JONGLIEREN MIT VARIABLEN

Lösung 4: $a) + b) + c)$

```
// Deklaration (a)
int x, y;

// Eingabe (b)
x = IOTools.readInt("Eingabe_x:_");
y = IOTools.readInt("Eingabe_y:_");

// Ausgabe (c)
System.out.println("x:_ " + x + " _-y:_ " + y);
```

Lösung 4: d)

```
// Tauschen
```

Lösung 4: d)

```
// Tauschen (mit Hilfsvariable)  
int z = x;
```

Lösung 4: d)

```
// Tauschen (mit Hilfsvariable)
int z = x;
x = y;
```

Lösung 4: d)

```
// Tauschen (mit Hilfsvariable)
int z = x;
x = y;
y = z;
```

Lösung 4: d)

```
// Tauschen (mit Hilfsvariable)
int z = x;
x = y;
y = z;

// Ausgabe
System.out.println("x:_ " + x + " _-y:_ " + y);
```

Lösung 4: e)

Lösung 4: e)

```
// Vergleich
int largerValue;
if (x >= y)
    largerValue = x;
else
    largerValue = y;
```

Lösung 4: e)

```
// Vergleich
int largerValue;
if (x >= y)
    largerValue = x;
else
    largerValue = y;

System.out.println("Der größere Wert von beiden ist: " + ↵
    largerValue);
```

Lösung 4: e)

```
// Vergleich
int largerValue;
if (x >= y)
    largerValue = x;
else
    largerValue = y;

System.out.println("Der größere Wert von beiden ist: " + ↵
    largerValue);
```

Ja, das ist eigentlich nur der „nicht-kleinere“ Wert!

Lösung 4: d) + e), aber schöner!

```
// Tauschen
```

Lösung 4: d) + e), aber schöner!

```
// Tauschen (mit XOR)
```

Lösung 4: d) + e), aber schöner!

```
// Tauschen (mit XOR)  
x = x ^ y;
```

JONGLIEREN MIT VARIABLEN

Lösung 4: d) + e), aber schöner!

```
// Tauschen (mit XOR)
x = x ^ y;
y = x ^ y;
```

JONGLIEREN MIT VARIABLEN

Lösung 4: d) + e), aber schöner!

```
// Tauschen (mit XOR)
x = x ^ y;
y = x ^ y; // x ^ y ^ y = x
```


JONGLIEREN MIT VARIABLEN

Lösung 4: d) + e), aber schöner!

```
// Tauschen (mit XOR)
x = x ^ y;
y = x ^ y; // x ^ y ^ y = x
x = x ^ y;
```

JONGLIEREN MIT VARIABLEN

Lösung 4: d) + e), aber schöner!

```
// Tauschen (mit XOR)
x = x ^ y;
y = x ^ y; // x ^ y ^ y = x
x = x ^ y; // x ^ y ^ x = y
```

JONGLIEREN MIT VARIABLEN

Lösung 4: d) + e), aber schöner!

```
// Tauschen (mit XOR)
x = x ^ y;
y = x ^ y; // x ^ y ^ y = x
x = x ^ y; // x ^ y ^ x = y
```

```
// besserer Vergleich
int largerValue
```

JONGLIEREN MIT VARIABLEN

Lösung 4: d) + e), aber schöner!

```
// Tauschen (mit XOR)
x = x ^ y;
y = x ^ y; // x ^ y ^ y = x
x = x ^ y; // x ^ y ^ x = y

// besserer Vergleich
int largerValue = x;
```

JONGLIEREN MIT VARIABLEN

Lösung 4: d) + e), aber schöner!

```
// Tauschen (mit XOR)
x = x ^ y;
y = x ^ y; // x ^ y ^ y = x
x = x ^ y; // x ^ y ^ x = y
```

```
// besserer Vergleich
int largerValue = x;
if (y > x) largerValue = y;
```

JONGLIEREN MIT VARIABLEN

Lösung 4: d) + e), aber schöner!

```
// Tauschen (mit XOR)
```

```
x = x ^ y;
```

```
y = x ^ y; // x ^ y ^ y = x
```

```
x = x ^ y; // x ^ y ^ x = y
```

```
// besserer Vergleich
```

```
int largerValue = x;
```

```
if (y > x) largerValue = y;
```

```
// oder direkt mit ternärem Operator
```

```
System.out.println("Der größere Wert von beiden ist: " + (x >= y) ? x : y);
```

Ich muss noch ein wenig schimpfen...



Warum ist der folgende Code schlecht?

```
negate :: Bool -> Bool
negate b
  | (b == True)   = False
  | (b == False) = True
```


TRUE-VERGLEICHE

Warum ist der folgende Code schlecht?

```
negate :: Bool -> Bool
negate b
  | (b == True)   = False
  | (b == False)  = True
```

Prüft Booleans

TRUE-VERGLEICHE

Warum ist der folgende Code schlecht?

```
negate :: Bool -> Bool
negate b
  | (b == True)   = False
  | (b == False)  = True
```

Prüft Booleans

The diagram illustrates the logic of the `negate` function. It shows how the value of `b` is compared to `True` and `False`. The first guard `(b == True)` evaluates to `False` when `b` is `True`, resulting in the expression `(True == True)`. The second guard `(b == False)` evaluates to `True` when `b` is `False`, resulting in the expression `(True == False)`. The text "Prüft Booleans" is at the bottom left.

- Wir vergleichen Wahrheitswerte

TRUE-VERGLEICHE

Warum ist der folgende Code schlecht?

```
negate :: Bool -> Bool
negate b
  | (b == True)   = False
  | (b == False)  = True
```

Prüft Booleans

$(True == True) = True$
 $(True == False) = False$

- Wir vergleichen Wahrheitswerte und erhalten Wahrheitswerte.

TRUE-VERGLEICHE

Warum ist der folgende Code schlecht?

```
negate :: Bool -> Bool
negate b
  | (b == True)   = False
  | (b == False)  = True
```

Prüft Booleans

Diagram illustrating the evaluation of the `negate` function:

- The function signature is `negate :: Bool -> Bool`.
- The function is called with argument `b`.
- The first guard `(b == True) = False` is evaluated. An arrow points from `True` to the result `True` above the function signature.
- The second guard `(b == False) = True` is evaluated. An arrow points from `True` to the result `True` above the function signature.
- The final result is `True`.

- Wir vergleichen Wahrheitswerte und erhalten Wahrheitswerte.
 - › Boole'sche Operatoren

TRUE-VERGLEICHE

Warum ist der folgende Code schlecht?

```
negate :: Bool -> Bool
negate b
  | (b == True)   = False
  | (b == False)  = True
  Prüft Booleans
```

Diagram illustrating the evaluation of the `negate` function:

- The function signature is `negate :: Bool -> Bool`.
- The function takes a parameter `b`.
- The function body consists of two cases:
 - `| (b == True) = False`: This case evaluates to `False` when `b` is `True`. The result is `False`.
 - `| (b == False) = True`: This case evaluates to `True` when `b` is `False`. The result is `True`.
- The text "Prüft Booleans" (Checks Booleans) is written below the function body.

- Wir vergleichen Wahrheitswerte und erhalten Wahrheitswerte.
 - › Boole'sche Operatoren

Wie können wir den Code verbessern?

```
negate :: Bool -> Bool
negate b
  | (b)      = False
  | (not b) = True
```

- Wir vergleichen Wahrheitswerte und erhalten Wahrheitswerte.
 - › Boole'sche Operatoren

BEDINGUNGEN UND ALTERNATIVEN

Wie können wir den Code verbessern?

```
negate :: Bool -> Bool
```

```
negate b
```

```
  | (b)      = False
```

```
  | (not b)  = True
```

 b ist nicht **True**, also bleibt uns nur noch ein Fall

- Wir vergleichen Wahrheitswerte und erhalten Wahrheitswerte.
 - **Boole'sche Operatoren**
- Wir können redundante Vergleiche reduzieren.
 - **otherwise (oder else)**

(REDUNDANTE) BOOLEAN-RÜCKGABE

Warum ist das immernoch nicht perfekt?


```
negate :: Bool -> Bool
negate b
  | b          = False
  | otherwise = True
```

- Wir vergleichen Wahrheitswerte und erhalten Wahrheitswerte.
 - Boole'sche Operatoren
- Wir können redundante Vergleiche reduzieren.
 - otherwise (oder else)

(REDUNDANTE) BOOLEAN-RÜCKGABE

Warum ist das immernoch nicht perfekt?

```
negate :: Bool -> Bool
negate b
  | b           = False
  | otherwise = True
```



- Wir vergleichen Wahrheitswerte und erhalten Wahrheitswerte.
 - Boole'sche Operatoren
- Wir können redundante Vergleiche reduzieren.
 - otherwise (oder else)

(REDUNDANTE) BOOLEAN-RÜCKGABE

Warum ist das immernoch nicht perfekt?

```
negate :: Bool -> Bool
```

```
negate b
```

```
  | b          = False
```

```
  | otherwise = True
```

Wir erinnern uns:

```
  | (b)          == True = False
```

```
  | (not b)      == True = True
```

- Wir vergleichen Wahrheitswerte und erhalten Wahrheitswerte.
 - Boole'sche Operatoren
- Wir können redundante Vergleiche reduzieren.
 - otherwise (oder else)
- Wir geben Wahrheitswerte explizit zurück.

(REDUNDANTE) BOOLEAN-RÜCKGABE

Warum ist das immernoch nicht perfekt?

```
negate :: Bool -> Bool
```

```
negate b
```

```
  | b          = False
```

```
  | otherwise = True
```

Wir erinnern uns:

```
  | (b)          == True = not (b) -- False
```

```
  | (not b)      == True = True
```

- Wir vergleichen Wahrheitswerte und erhalten Wahrheitswerte.
 - Boole'sche Operatoren
- Wir können redundante Vergleiche reduzieren.
 - otherwise (oder else)
- Wir geben Wahrheitswerte explizit zurück.

(REDUNDANTE) BOOLEAN-RÜCKGABE

Warum ist das immernoch nicht perfekt?

```
negate :: Bool -> Bool
```

```
negate b
```

```
  | b           = False
```

```
  | otherwise = True
```

Wir erinnern uns:

```
  | (b)           == True = not (b) -- False
```

```
  | (not b)       == True = (not b) -- True
```

- Wir vergleichen Wahrheitswerte und erhalten Wahrheitswerte.
 - Boole'sche Operatoren
- Wir können redundante Vergleiche reduzieren.
 - otherwise (oder else)
- Wir geben Wahrheitswerte explizit zurück.
 - Boole'sche Operatoren

VERGLEICH

Schlechter Code

```
negate :: Bool -> Bool
negate b
  | (b == True)   = False
  | (b == False) = True
```

Verbesserter Code

```
negate :: Bool -> Bool
negate b = not b
```

Die Folien gibts eigentlich nur,
weil ich die Musterlösung unübersichtlich finde xD

Name Last

Münster, 13. Januar 2025

name.last@uni-muenster.de