Dr. Annette Bieniusa
Mathias Weber, M. Sc.
Peter Zeller, M. Sc.

TU Kaiserslautern
Fachbereich Informatik
AG Softwaretechnik

Lösungshinweise/-vorschläge zum Übungsblatt 14: Software-Entwicklung 1 (WS 2017/18)

Die Hinweise und Vorschläge in diesem Dokument sollen der Lösungsfindung dienen und erheben demnach weder Anspruch auf Vollständigkeit noch Korrektheit. Sollten Sie Fehler finden, würden wir uns freuen, wenn Sie uns diese mitteilen. (Kontaktdaten finden Sie auf unserer Webpräsenz.)

Aufgabe 1 Lambda-Kalkül: Freie Variablen (9 Punkte)

Verwenden Sie die formale Definition für freie Variablen und berechnen Sie damit die Menge der freien Variablen in folgenden Lambda-Termen. Geben Sie dabei alle Zwischenschritte an.

a) `(λx → g x) x`

Detaillierte Rechnung:

$$
\begin{aligned}
& FV(\texttt{(}\lambda\texttt{x} \rightarrow \texttt{g x)}\ \texttt{x}) \\
= \ & FV(\texttt{(}\lambda\texttt{x} \rightarrow \texttt{g x)}) \cup FV(\texttt{x}) \\
= \ & (FV(\texttt{g x}) \setminus \{x\}) \cup FV(\texttt{x}) \\
= \ & ((FV(\texttt{g}) \cup FV(\texttt{x})) \setminus \{x\}) \cup FV(\texttt{x}) \\
= \ & ((\{g\} \cup FV(\texttt{x})) \setminus \{x\}) \cup FV(\texttt{x}) \\
= \ & ((\{g\} \cup \{x\}) \setminus \{x\}) \cup FV(\texttt{x}) \\
= \ & ((\{g, x\}) \setminus \{x\}) \cup FV(\texttt{x}) \\
= \ & \{g\} \cup FV(\texttt{x}) \\
= \ & \{g\} \cup \{x\} \\
= \ & \{g, x\}
\end{aligned}
$$

Es dürfen hier auch mehrere Teilterme in einer Zeile vereinfacht werden:

$$
\begin{aligned}
& FV(\texttt{(}\lambda\texttt{x} \rightarrow \texttt{g x)}\ \texttt{x}) \\
= \ & FV(\texttt{(}\lambda\texttt{x} \rightarrow \texttt{g x)}) \cup FV(\texttt{x}) \\
= \ & (FV(\texttt{g x}) \setminus \{x\}) \cup \{x\} \\
= \ & ((FV(\texttt{g}) \cup FV(\texttt{x})) \setminus \{x\}) \cup \{x\} \\
= \ & ((\{g\} \cup \{x\}) \setminus \{x\}) \cup \{x\} \\
= \ & ((\{g, x\}) \setminus \{x\}) \cup \{x\} \\
= \ & \{g\} \cup \{x\} \\
= \ & \{g, x\}
\end{aligned}
$$

- Richtiges Ergebnis (1 Punkt)
- Alle Zwischenschritte korrekt (2 Punkte)

b) `(λx → (λy → y) y) x`

$$
\begin{aligned}
& FV(\texttt{(}\lambda\texttt{x} \rightarrow \texttt{(}\lambda\texttt{y} \rightarrow \texttt{y) y) x}) \\
= \; & FV(\lambda\texttt{x} \rightarrow \texttt{(}\lambda\texttt{y} \rightarrow \texttt{y) y}) \cup FV(\texttt{x}) \\
= \; & (FV(\texttt{(}\lambda\texttt{y} \rightarrow \texttt{y) y}) \setminus \{x\}) \cup \{x\} \\
= \; & ((FV(\lambda\texttt{y} \rightarrow \texttt{y}) \cup FV(\texttt{y})) \setminus \{x\}) \cup \{x\} \\
= \; & (((FV(\texttt{y}) \setminus \{y\}) \cup \{y\}) \setminus \{x\}) \cup \{x\} \\
= \; & ((((\{y\} \setminus \{y\}) \cup \{y\}) \setminus \{x\}) \cup \{x\} \\
= \; & \{x, y\}
\end{aligned}
$$

- Richtiges Ergebnis (1 Punkt)
- Alle Zwischenschritte korrekt (2 Punkte)

c) `f (λx → f (λy → f x y))`

$$
\begin{aligned}
& FV(\texttt{f (x}\rightarrow\texttt{ f (y}\rightarrow\texttt{ f x y))}) \\
= \; & FV(\texttt{f}) \cup FV(\texttt{x}\rightarrow\texttt{ f (y}\rightarrow\texttt{ f x y)}) \\
= \; & \{f\} \cup (FV(\texttt{f (y}\rightarrow\texttt{ f x y)}) \setminus \{x\}) \\
= \; & \{f\} \cup ((FV(\texttt{f}) \cup FV(\texttt{y}\rightarrow\texttt{ f x y})) \setminus \{x\}) \\
= \; & \{f\} \cup ((\{f\} \cup (FV(\texttt{f x y}) \setminus \{y\}) \setminus \{x\}) \\
= \; & \{f\} \cup ((\{f\} \cup ((FV(\texttt{f x}) \cup FV(\texttt{y})) \setminus \{y\}) \setminus \{x\}) \\
= \; & \{f\} \cup ((\{f\} \cup ((FV(\texttt{f}) \cup FV(\texttt{x}) \cup \{y\}) \setminus \{y\}) \setminus \{x\}) \\
= \; & \{f\} \cup ((\{f\} \cup ((\{f\} \cup \{x\} \cup \{y\}) \setminus \{y\}) \setminus \{x\}) \\
= \; & \{f\}
\end{aligned}
$$

- Richtiges Ergebnis (1 Punkt)
- Alle Zwischenschritte korrekt (2 Punkte)

Aufgabe 2 Lambda-Kalkül: Substitutionen (6 Punkte)

Sind die folgenden Substitutionen erlaubt? Falls nicht, begründen Sie Ihre Antwort. Falls ja, berechnen Sie für die Substitution das Ergebnis mit Hilfe der formalen Definition. Geben Sie dabei alle Zwischenschritte an.

a) `((f x) y)[y/x]`

```
  ((f x) y)[y/x]
= ((f x)[y/x] y[y/x])
= ((f[y/x] x[y/x]) y[y/x])
= ((f y) y)
```

- Richtiges Ergebnis (1 Punkt)

- Alle Zwischenschritte korrekt (1 Punkt)

b) `(λz → f (λx → (f x) z))[(g z)/f]`

Die Substitution ist nicht erlaubt, da z nach der Substitution gebunden wäre:

```
  (λz → f (λx → (f x) z))[(g z)/f]
= (λz → (f (λx → (f x) z))[(g z)/f])
= (λz → (f[(g z)/f] (λx → (f x) z)[(g z)/f]))
= (λz → ((g z) (λx → (f x) z)[(g z)/f]))
```

- Nicht erlaubt (1 Punkt)
- Begründung (zum Beispiel: ausrechnen, was wäre wenn sie erlaubt wäre) (1 Punkt)

c) `(x (λx → f x))[f/x]`

Hier ist zu beachten, dass das innere, gebundene Vorkommen von x nicht ersetzt wird.

```
  (x (λx → f x))[f/x]
= (x[f/x] (λx → f x)[f/x])
= (f (λx → f x)[f/x])
= (f (λx → f x))
```

- Richtiges Ergebnis (1 Punkt)
- Alle Zwischenschritte korrekt (1 Punkt)

Aufgabe 3 Lambda-Kalkül: β-Normalform (12 Punkte)

Bringen Sie die folgenden Lambda-Terme in eine β-Normalform. Geben Sie dabei alle Zwischenschritte an (die erforderlichen Substitution sollen dabei für β-Konversionen explizit angegeben werden, dürfen dann aber in einem Schritt ausgewertet werden; α-Konversionen dürfen in einem Schritt durchgeführt werden).

Die Lambda-Terme in dieser Aufgabe enthalten Zahlen und mathematische Operationen, welche mathematisch auszuwerten sind.

a) `(λx → x*6) ((λx → 2+x) 5)`

```
  (λx → x*6) ((λx → 2+x) 5)
= (λx → x*6) ((2+x)[5/x])
= (λx → x*6) (2+5)
= (λx → x*6) 7
= (x*6)[7/x]
= 7*6
= 42
```

- Richtiges Ergebnis (1 Punkt)
- Alle Zwischenschritte (2 Punkte)

b) `((λx → x 6) (λx y → x+y)) 5`

```
  ((λx → x 6) (λx y → x+y)) 5
= ((x 6)[(λx y → x+y)/x]) 5
= ((λx y → x+y) 6)  5
= ((λy → x+y)[6/x])  5
```

```
= (λy → 6+y)  5
= (6+y)[5/y]
= 6+5
= 11
```

- Richtiges Ergebnis (1 Punkt)
- Alle Zwischenschritte (2 Punkte)

c) `(λx → ((λy → y*y) x))`

```
  (λx → ((λy → y*y) x))
= (λx → ((y*y)[x/y]))
= (λx → x*x)
```

- Richtiges Ergebnis (1 Punkt)
- Alle Zwischenschritte (1 Punkt)

d) `((λf y → f (f y)) (λx → y + x)) y`

```
  ((λf y → f (f y)) (λx → y + x)) y
= ((λf z → f (f z)) (λx → y + x)) y
= ((λz → f (f z))[(λx → y + x)/f]) y
= (λz → (λx → y + x) ((λx → y + x) z)) y
= ((λx → y + x) ((λx → y + x) z))[y/z]
= (λx → y + x) ((λx → y + x) y)
= (λx → y + x) ((y + x)[y/x])
= (λx → y + x) (y + y)
= (y + x)[(y + y)/x]
= (y + (y + y))
```

- Richtiges Ergebnis (1 Punkt)
- Alle Zwischenschritte (3 Punkte)

Aufgabe 4 Lambda-Kalkül: Auswertungsstrategien (10 Punkte)

Werten Sie die folgenden Lambda-Terme jeweils mit den Auswertungsstrategien call-by-value und call-by-name aus. Geben Sie dabei alle Zwischenschritte an (ohne Angabe der Substitutionen).

a) `(λx y → y) ((λx → 5*x) 6) 4`

Call-by-value:

```
  (λx y → y) ((λx → 5*x) 6) 4
= (λx y → y) (5*6) 4
= (λx y → y) 30 4
= (λy → y) 4
= 4
```

Call-by-name:

```
  (λx y → y) ((λx → 5*x) 6) 4
= (λy → y) 4
= 4
```

Hier erfordert Call-by-name weniger Schritte, da der nicht benötigte erste Parameter nicht ausgewertet werden muss.

- Auswertung call-by-value (2 Punkte)
- Auswertung call-by-name (1 Punkt)

b) `(λx y → x+x) ((λx → 5*x) 6) 4`

Call-by-value:

```
  (λx y → x+x) ((λx → 5*x) 6) 4
= (λx y → x+x) (5*6) 4
= (λx y → x+x) 30 4
= (λy → 30+30) 4
= 30+30
= 60
```

Call-by-name:

```
  (λx y → x+x) ((λx → 5*x) 6) 4
= (λy → ((λx → 5*x) 6) + ((λx → 5*x) 6)) 4
= ((λx → 5*x) 6) + ((λx → 5*x) 6))
= (5*6 + ((λx → 5*x) 6))
= (30 + ((λx → 5*x) 6))
= (30 + 5*6)
= (30 + 30)
= 60
```

Hier erfordert Call-by-value weniger Schritte, da der erste Parameter nur einmal ausgewertet werden muss.

- Auswertung call-by-value (2 Punkte)
- Auswertung call-by-name (2 Punkte)

c) `(λx → z) ((λx → x x x) (λx → x x x))`

Call-by-value:

```
  (λx → z) ((λx → x x x) (λx → x x x))
= ((λx → z) ((λx → ((x x) x)) (λx → ((x x) x))))
= ((λx → z) (((λx → ((x x) x)) (λx → ((x x) x))) (λx → ((x x) x))))
= ((λx → z) ((((λx → ((x x) x)) (λx → ((x x) x))) (λx → ((x x) x))) (λx → ((x x)
    x))))
= ((λx → z) (((((λx → ((x x) x)) (λx → ((x x) x))) (λx → ((x x) x))) (λx → ((x x)
     x))) (λx → ((x x) x))))
= ((λx → z) ((((((λx → ((x x) x)) (λx → ((x x) x))) (λx → ((x x) x))) (λx → ((x x
    ) x))) (λx → ((x x) x))) (λx → ((x x) x))))
= ((λx → z) (((((((λx → ((x x) x)) (λx → ((x x) x))) (λx → ((x x) x))) (λx → ((x
    x) x))) (λx → ((x x) x))) (λx → ((x x) x))) (λx → ((x x) x))))
= ((λx → z) ((((((((λx → ((x x) x)) (λx → ((x x) x))) (λx → ((x x) x))) (λx → ((x
     x) x))) (λx → ((x x) x))) (λx → ((x x) x))) (λx → ((x x) x))) (λx → ((x x) x)
    )))
= ...
```

Call-by-name:

```
  (λx → z) ((λx → x x x) (λx → x x x))
= z
```

Die Ausführung mit Call-by-value liefert immer größer werdende Terme. Die Auswertung mit Call-

by-name terminiert hingegen in einem Schritt, weil der Parameter nicht verwendet wird.

- Auswertung call-by-value (2 Punkte)
- Auswertung call-by-name (1 Punkt)

Aufgabe 5 (Zusatzaufgabe) Boolsche Werte im Lambda-Kalkül

Die Zusatzaufgaben auf diesem Blatt gehen über den Stoff von Vorlesung und Klausur hinaus. Sie können die Lösung zu diesen Aufgaben im Exclaim in der Zusatzübung SE1WS17Z abgeben.

Wir definieren die Konstanten `true` und `false` wie folgt:

```
true  = (λx y → x)
false = (λx y → y)
```

Verwenden Sie diese Definitionen, um die folgenden Funktionen zu definieren:

a) Eine Funktion `not`, welche einen Boolschen Wert nimmt und seine Negation berechnet.

```
not true  = false
not false = true

not = (λp → p false true)
```

Oder:

```
not = (λp t f → p f t)
```

b) Eine Funktion `and`, welche zwei Boolsche Werte nimmt und `true` liefert, genau dann wenn beide Parameter `true` sind.

```
and true  true  = true
and true  false = false
and false true  = false
and false false = false

and = (λp q t f → p (q t f) f)
```

Oder:

```
and = (λp q → p q p)
```

Aufgabe 6 (Zusatzaufgabe) Java Lambdas und Streams

Sie können die Lösung zu dieser Aufgabe im Exclaim in der Zusatzübung SE1WS17Z abgeben.

a) Schreiben Sie eine Methode `static List<String> getMealNames(Restaurant r)`, welche die Namen der Mahlzeiten zurückgibt, welche das Restaurant `r` auf der Speisekarte hat.

```
static List<String> getMealNames(Restaurant r) {
    return r.getMenu().stream()
            .map(m → m.getName())
            .collect(Collectors.toList());
}
```

b) Schreiben Sie eine Methode `static List<Meal> getVegetarianMeals(Restaurant r)`, welche die vegetarischen Mahlzeiten liefert, die Restaurant `r` auf der Speisekarte hat.

```
static List<Meal> getVegetarianMeals(Restaurant r) {
    return r.getMenu().stream()
            .filter(m → m.isVegetarian())
            .collect(Collectors.toList());
}
```

c) Schreiben Sie eine Methode `static List<Restaurant> searchRestaurants(List<Restaurant> rs, String search)`, welche eine Liste von Restaurants `rs` und einen Suchstring `search` nimmt und alle Restaurants liefert, die eine Mahlzeit auf dem Speiseplan hat, deren Name den Suchstring `search` enthält.

```
static List<Restaurant> searchRestaurants(List<Restaurant> rs, String search) {
    return rs.stream()
            .filter(r →
                    r.getMenu().stream()
                            .anyMatch(m → m.getName().contains(search)))
            .collect(Collectors.toList());
}
```

d) Schreiben Sie eine Methode `static void sort(List<Meal> meals)`, welche eine Liste von Mahlzeiten nimmt und diese zuerst nach ihrem Preis sortiert (billigste zuerst) und bei gleichem Preis nach ihrer Bewertung (höchste zuerst).

```
static void sort(List<Meal> meals) {
    Collections.sort(meals,
            Comparator.comparing(Meal::getPrice)
                    .thenComparing(Meal::getRating, Collections.reverseOrder()));
}
```

Aufgabe 7 (Zusatzaufgabe) Lambda-Kalkül in Java

Sie können die Lösung zu dieser Aufgabe im Exclaim in der Zusatzübung SE1WS17Z abgeben.

a) Implementieren Sie eine Methode `Set<Variable> variables()` in der Klasse `Term`, welche die Variablen zurückgibt, die im Term vorkommen.

Wir verwenden hier dynamische Methodenbindung. Daher lassen wir die Methode in der Klasse `Term` abstrakt und Implementieren die Methode in den Subklassen.

In der Klasse `Variable`:

```
@Override
public Set<Variable> variables() {
    HashSet<Variable> res = new HashSet<>();
    res.add(this);
    return res;
}
```

In der Klasse `FunctionApplication`:

```
@Override
public Set<Variable> variables() {
    Set<Variable> variables = function.variables();
    variables.addAll(argument.variables());
    return variables;
}
```

In der Klasse `FunctionAbstraction`:

```
@Override
public Set<Variable> freeVariables() {
    Set<Variable> free = body.freeVariables();
    free.remove(variable);
    return free;
}
```

b) Implementieren Sie eine Methode `Set<Variable> freeVariables()` in der Klasse `Term`, welche die **freien** Variablen zurückgibt, die im Term vorkommen.

In der Klasse `Variable`:

```
@Override
public Set<Variable> freeVariables() {
    return variables();
}
```

In der Klasse `FunctionApplication`:

```
@Override
public Set<Variable> freeVariables() {
    Set<Variable> freeVariables = function.freeVariables();
    freeVariables.addAll(argument.freeVariables());
    return freeVariables;
}
```

In der Klasse `FunctionAbstraction`:

```
@Override
public Set<Variable> variables() {
    Set<Variable> variables = body.variables();
    variables.add(variable);
    return variables;
}
```

c) Implementieren Sie eine Methode `Term substitute(Term t, Variable x)`, welche alle freien Variablen `x` im aktuellen Term durch den Term `t` ersetzt. Falls die Substitution nicht erlaubt ist, soll die Methode entsprechende α-Konversionen ausführen um die Substitution zu ermöglichen.

In der Klasse `Variable`:

```
@Override
public Term substitute(Term t, Variable x) {
    if (x.equals(this)) {
        return t;
    } else {
        return this;
    }
}
```

In der Klasse `FunctionApplication`:

```
@Override
public Term substitute(Term t, Variable x) {
    return new FunctionApplication(function.substitute(t, x), argument.
substitute(t, x));
}
```

In der Klasse `FunctionAbstraction`:

```
@Override
public Term substitute(Term t, Variable x) {
    if (x.equals(variable)) {
        return this;
    } else {
        Set<Variable> freeVars = t.freeVariables();
        if (freeVars.contains(variable)) {
            Set<Variable> bodyVars = body.variables();
            for (int i = 1; ; i++) {
                Variable newVar = new Variable(variable.getName() + i);
                if (!freeVars.contains(newVar) && !bodyVars.contains(newVar)) {
                    Term newBody = body.substitute(newVar, variable).substitute(t, x);
                    return new FunctionAbstraction(newVar, newBody);
                }
            }
        } else {
            return new FunctionAbstraction(variable, body.substitute(t, x));
        }
    }
}
```

d) Implementieren Sie die Methode `Term evaluateStep(Term t)` in der vorgegebenen Klasse `EvaluateCallByValue`. Diese Methode soll einen Berechnungsschritt auf dem Term `t` mit der call-by-value Strategie ausführen und den umgeformten Term zurückgeben.

```
public class EvaluateCallByValue extends Evaluator {
    public Term evaluateStep(Term t) {
        if (t instanceof FunctionApplication) {
            FunctionApplication app = (FunctionApplication) t;

            if (app.getArgument() instanceof FunctionApplication) {
                // Form (_ (f x))
                // erst Parameter auswerten:
                return new FunctionApplication(
                        app.getFunction(),
                        evaluateStep(app.getArgument()));
            } else if (app.getFunction() instanceof FunctionApplication) {
                // Form ((f x) _)
                // Funktion auswerten:
                return new FunctionApplication(
                        evaluateStep(app.getFunction()),
                        app.getArgument());
            } else if (app.getFunction() instanceof FunctionAbstraction) {
                // Form ((λx → body) t)
                FunctionAbstraction f = (FunctionAbstraction) app.getFunction();

                // Aufruf auswerten
                return f.getBody().substitute(app.getArgument(), f.getVariable());
            }
        }
        return t;
    }
}
```

e) Implementieren Sie die Methode `Term evaluateStep(Term t)` in der vorgegebenen Klasse `EvaluateCallByName`. Diese Methode soll einen Berechnungsschritt auf dem Term `t` mit der call-by-name Strategie ausführen und den umgeformten Term zurückgeben.

```
public class EvaluateCallByName extends Evaluator {
    public Term evaluateStep(Term t) {
        if (t instanceof FunctionApplication) {
            FunctionApplication app = (FunctionApplication) t;

            if (app.getFunction() instanceof FunctionApplication) {
                // Form ((f x) _)
                // Funktion auswerten:
                return new FunctionApplication(
                        evaluateStep(app.getFunction()),
                        app.getArgument());
            } else if (app.getFunction() instanceof FunctionAbstraction) {
                // Form ((λx → body) t)
                FunctionAbstraction f = (FunctionAbstraction) app.getFunction();

                // Aufruf auswerten
                return f.getBody().substitute(app.getArgument(), f.getVariable());
            }
        }
        return t;
    }
}
```

Aufgabe 8 (Zusatzaufgabe) Racket

Sie können die Lösung zu dieser Aufgabe im Exclaim in der Zusatzübung SE1WS17Z abgeben.

a) Die Funktion `laenge` berechnet die Länge einer Liste.

```
(define (laenge l)
  (if (empty? l)
      0
      (+ 1 (laenge (rest l)))))
```

b) Die Funktion `nimm` nimmt eine ganze Zahl n und eine Liste l und gibt eine Liste der ersten n Elemente von l zurück.

```
(define (nimm n l)
  (if (or (<= n 0) (empty? l))
      empty
      (cons (first l) (nimm (- n 1) (rest l)))))
```

c) Die Funktion `werfWeg` nimmt eine ganze Zahl n und eine Liste l und gibt eine Liste ohne die ersten n Elemente von l zurück.

```
(define (werfWeg n l)
  (if (<= n 0)
      l
      (werfWeg (- n 1) (rest l))))
```

d) Die Funktion `startetMit` nimmt eine Liste x und eine Liste l und prüft, ob die Liste x ein Prefix der Liste l ist.

```
(define (startetMit x l)
  (or
   (empty? x)
   (and
    (not (empty? l))
    (= (first x) (first l))
    (startetMit (rest x) (rest l)))))
```

e) Die Funktion `verbinde` nimmt zwei Listen und hängt diese aneinander.

```
(define (verbinde x y)
  (if (empty? x)
      y
      (cons (first x) (verbinde (rest x) y))))
```

f) Die Funktion `ersetzeAlle` nimmt zwei Zahlen x und y und eine Liste l. Sie ersetzt alle Vorkommen von x in l durch y.

```
(define (ersetzeAlle x y l)
  (if (empty? l)
      empty
      (cons
       (if (= (first l) x)
           y
           (first l))
       (ersetzeAlle x y (rest l)))))

; alternative Definition mit map:
(define (ersetzeAlle2 x y l)
  (map (lambda (e) (if (= e x) y e)) l))
```

g) Die Funktion `ersetzeEins` nimmt zwei Zahlen x und y und eine Liste l. Sie ersetzt das erste Vorkommen von x in l durch y.

```
(define (ersetzeEins x y l)
  (if (empty? l)
      empty
      (cons
       (if (= (first l) x)
           y
           (first l))
       l)))
```

h) Die Funktion `ersetzeListe` nimmt zwei Listen x und y und eine Liste l. Sie ersetzt alle Vorkommen von x in l durch y. Der Unterschied zur Funktion `ersetzeAlle` ist, dass hier ganze Teillisten ersetzt werden und nicht nur einzelne Einträge in der Liste.

```
(define (ersetzeListe x y l)
  (if (empty? l)
      empty
      (if (startetMit x l)
          (verbinde y (ersetzeListe x y (werfWeg (laenge x) l)))
          (cons (first l) (ersetzeListe x y (rest l))))))
```

i) Die Funktion `maxList` sucht die größte Zahl aus einer Liste von Zahlen heraus.

```
(define (max x y)
  (if (< x y) y x))

(define (maxList l)
  (if (empty? l)
      0
      (max (first l) (maxList (rest l)))))

; Alternative mit foldl
(define (maxList2 l)
  (foldl max 0 l))
```

j) Die Funktion `element` nimmt eine Zahl und eine Liste und prüft, ob die gegebene Zahl in der gegebenen Liste vorkommt.

```
(define (element x l)
  (and
   (not (empty? l))
   (or
    (= x (first l))
    (element x (rest l)))))
```