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

TU Kaiserslautern
Fachbereich Informatik
AG Softwaretechnik

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

Ausgabe: 05.02.18
Abgabe: 12.02.18

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`

b) `(λx → (λy → y) y) x`

c) `f (λx → f (λy → f x y))`

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]`

b) `(λz → f (λx → (f x) z))[(g z)/f]`

c) `(x (λx → f x))[f/x]`

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 explizit angegeben werden, dürfen dann aber in einem Schritt ausgewertet werden).

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

a) `(λx → x*6) ((λx → 2+x) 5)`

b) `((λx → x 6) (λx y → x+y)) 5`

c) `(λx → ((λy → y*y) x))`

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

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`

b) `(λx y → x+x) ((λx → 5*x) 6) 4`

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

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
```

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
```

Aufgabe 6 (Zusatzaufgabe) Java Lambdas und Streams

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

Gegeben sind die folgenden Klassen, welche Restaurants und deren Angebote an Speisen modellieren:

```
class Restaurant {
    private final String name;
    private final List<Meal> menu;

    Restaurant(String name, List<Meal> menu) {
        this.name = name;
        this.menu = menu;
    }

    String getName() {
        return name;
    }

    List<Meal> getMenu() {
        return menu;
    }

}
```

```
class Meal {
    private final String name;
    private final boolean isVegetarian;
    private final double price;
    private double rating;

    Meal(String name, boolean isVegetarian,
    double price, double rating) {
        this.name = name;
        this.isVegetarian = isVegetarian;
        this.price = price;
        this.rating = rating;
    }

    String getName() {
        return name;
    }

    boolean isVegetarian() {
        return isVegetarian;
    }

    double getPrice() {
        return price;
    }

    double getRating() {
        return rating;
    }
}
```

Verwenden Sie für diese Aufgabe die Java Stream API, insbesondere:

- Die Methode `stream` aus dem Interface `java.util.Collection`.
- Die Methoden `map`, `filter`, `collect`, `anyMatch` aus dem Interface `java.util.stream.Stream`.
- Die statische Methode `toList` aus der Klasse `java.util.stream.Collectors`.
- Die statischen Methoden `sort` und `reverseOrder` aus der Klasse `java.util.Collections`.
- Die statische Methode `comparing` und die Methode `thenComparing` aus dem Interface `java.util.Comparator`.

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.

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

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.

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).

Aufgabe 7 (Zusatzaufgabe) Lambda-Kalkül in Java

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

Ziel dieser Aufgabe ist es ein Java-Programm zu schreiben, welches Lambda-Terme auswerten kann. Dazu können Lambda-Terme in Java durch Klassen dargestellt werden, welche die Terme in einer baumartigen Struktur (Syntax-Baum) darstellen:

```
public abstract class Term {

}

public class FunctionApplication extends Term {
    private final Term function;
    private final Term argument;

    public FunctionApplication(Term function,
    Term argument) {
        this.function = function;
        this.argument = argument;
    }

    public Term getFunction() {
        return function;
    }

    public Term getArgument() {
        return argument;
    }
}
```

```
public class Variable extends Term {
    private final String name;

    public Variable(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
public class FunctionAbstraction extends Term {
    private final Variable variable;
    private final Term body;

    public FunctionAbstraction(Variable
    variable, Term body) {
        this.variable = variable;
        this.body = body;
    }

    public Variable getVariable() {
        return variable;
    }

    public Term getBody() {
        return body;
    }
}
```

Laden Sie sich zum Lösen der Aufgabe die Vorlage `lambda.zip` herunter, welche auch einen Parser enthält, der aus einer String-Darstellung eines Lambda-Terms die entsprechenden Klassen aufbaut. Die Klasse `LambdaTest` enthält Testfälle für die geforderten Methoden.

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

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

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.

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.

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.

Aufgabe 8 (Zusatzaufgabe) Racket

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

Racket ist eine funktionale Programmiersprache, mit der Sie Ihr Wissen über den Lambda-Kalkül praktisch anwenden können. Einen Download und weitere Informationen zur Sprache finden Sie unter https://racket-lang.org/. Auf den Rechnern tux5 und tux6 des SCI ist Racket bereits für Sie installiert.

Nachdem Sie Den Racket-Editor DrRacket mit dem Befehl `drracket` gestartet haben, können Sie im Menü die Sprache auswählen. Wählen Sie dort den ersten Eintrag (Racket) als Sprache. Anschließend können Sie entweder Definition in den oberen Bereich schreiben oder im unteren Bereich Ausdrücke auswerten.

Die Syntax der grundlegenden Lambda-Terme unterscheidet sich in Racket kaum von der aus der Vorlesung bekannten Syntax.

	Vorlesung	Racket
Variablen	`x`	`x`
Funktionsaufruf	`(T T)`	`(T T)`
Abstraktion	`(λx → T)`	`(lambda (x) T)`

Des weiteren bietet Racket einige weitere Elemente wie Zahlen und die Funktion `+` und andere mathematische Operatoren (`+`, `-`, `*`, `/`, `=`, `<`, `<=`, `>=` , `>`). Dabei werden Operatoren wie `+` wie alle Funktionen auch vor und nicht zwischen die Parameter geschrieben, zum Beispiel können Sie die folgenden Terme in den unteren Eingabebereich von DrRacket eingeben:

```
> (+ 2 3)
5
> ((lambda (x) (+ x 1)) 5)
6
```

Für Wahrheitswerte bietet Racket die Konstanten `#t` für wahr und `#f` für falsch. Außerdem können die Operatoren **`if`**, **`and`**, **`or`**, **`not`** mit Wahrheitswerten verwendete werden:

```
> (and (> 10 5) (not (> 5 10)))
#t
> (if (< 5 10) (+ 1 2 3) (* 4 5))
6
```

Racket erlaubt es auch eine Funktionsabstraktion mit mehreren Parametern zu definieren. Im Gegensatz zum Lambda-Kalkül ist eine Funktion mit 2 Parametern aber nicht äquivalent zu einer einstelligen Funktion, welche eine Funktion zurückgibt.

```
> ((lambda (x y) (+ x y)) 1 2)
3
```

Mit der eingebauten Anweisung **`define`** lassen sich Abkürzungen definieren. Das folgende Beispiel definiert `f` als die Funktion, die ihren Parameter verdoppelt.

```
> (define f (lambda (x) (* 2 x)))
> (f 4)
8
```

Bei Definitionen lässt sich der Parameter auch direkt ohne **`lambda`**-Ausdruck angeben.

```
> (define (f x) (* 2 x))
```

Auch rekursive Definitionen sind erlaubt:

```
> (define (f x)
    (if (= x 0)
        1
        (* x (f (- x 1)))))
> (f 5)
120
```

Racket enthält außerdem `let`-Ausdrücke, welche wie die `let`-Ausdrücke aus Vorlesung funktionieren. Dabei können in Racket auch mehrere Variablen mit einem `let` gebunden werden:

```
> (let ((x 4)) (+ x 1))
5
> (let ((x 4) (y 5)) (+ x y))
9
```

Für rekursive Bindungen kann `letrec` statt `rec` verwendet werden.

```
> (letrec
      ((f (lambda (x)
            (if (= x 0)
                1
                (* x (f (- x 1)))))))
    (f 5))
120
```

Racket bietet auch Unterstützung für Listen. Die wichtigsten Funktionen sind hierbei:

- `empty` - Erstellt eine leere Liste
- `cons x l` - Erstellt eine Liste mit `x` als erstes Element und `l` als Rest der Liste.
- `first l` - Liefert das erste Element einer Liste `l`
- `rest l` - Liefert die Liste `l` ohne das erste Element
- `empty? l` - Ergibt `#t`, wenn die Liste `l` leer ist, sonst `#f`

Außerdem lassen sich Listen mit der Notation `(list ...)` erstellen, wobei nach `list` eine beliebige Zahl von Elementen steht.

```
> (list 1 2 3)
'(1 2 3)
> (first (list 1 2 3))
1
> (rest (list 1 2 3))
'(2 3)
> (empty? (list 1 2 3))
#f

; Kommentare beginnen in Racket uebrigens mit einem Semikolon.
```

Nach dieser kurzen Einführung sollten Sie in der Lage sein, die Beispiele aus der Vorlesung nach Racket zu übersetzen. Weitere Dokumentation zu Racket finden Sie unter https://docs.racket-lang.org/. Um das Programmieren mit Racket zu üben, können Sie sich an den folgenden Aufgaben versuchen:

Viele der geforderten Funktionen sind bereits in der Standardbibliothek vorhanden. Verwenden Sie zum Üben möglichst nur die grundlegenden Listenfunktionen, die auf diesem Blatt gezeigt wurden.

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

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

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.

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

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

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

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.

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.

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

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