# Formale Grundlagen der Informatik 3

Prof. Dr. Christoph Walther / Visar Januzaj, Nathan Wasser
Technische Universität Darmstadt — Wintersemester 2011/12

## Praktische Übung 1

---

**Version 1 vom 28.10.2011**

Diese Übung wird in der **KW 46 (14.–18.11.2011)** testiert. Reichen Sie dazu Ihre mit ✓eriFun 3.4 erstellte `.vf`-Lösungsdatei bis spätestens **Sonntag, 13.11.2011 23:59 Uhr** online im Kursportal ein. Später eintreffende Lösungen können nicht akzeptiert werden.

Bitte reservieren Sie bis zur oben genannten Frist im Kursportal einen Testattermin für die Testatwoche, zu dem alle Mitglieder Ihrer Praktikumsgruppe erscheinen können. Das Erscheinen zum vereinbarten Testattermin ist eine notwendige Voraussetzung, um das Testat zu bekommen.

Bei Fragen und Problemen können Sie neben den Poolbetreuungszeiten der Tutoren auch das Forum zur Veranstaltung nutzen.

Der Fachbereich Informatik misst der Einhaltung der Grundregeln der wissenschaftlichen Ethik großen Wert bei. Zu diesen gehört auch die strikte Verfolgung von Plagiarismus. Mit der Abgabe einer Lösung bestätigen Sie, dass Sie der alleinige Autor des gesamten Materials sind. Bei Unklarheiten zu diesem Thema finden Sie weiterführende Informationen unter
`http://www.informatik.tu-darmstadt.de/Plagiarism`
oder sprechen Sie Ihren Betreuer an.

## Vorbemerkung

In dieser Praxisübung werden Sie mit der Bedienung von ✓eriFun vertraut gemacht. Sie werden Implementieren, Testen, Widerlegen und Beweisen. Oft werden Sie Hilfslemmata selbst erfinden müssen. Hierzu dient das in der Vorlesung behandelte Verfahren der *Generalisierung*. Machen Sie sich noch einmal mit dem Begriff vertraut, bevor Sie die folgenden Aufgaben bearbeiten.

Bauen Sie bei der Lösung der nachfolgenden Aufgaben auf der Datei `Praxis1.vf` auf, die bereits einige grundlegende Definitionen enthält.

## Praktische Übung 1.1 (Testen und Verifizieren)

In dieser Aufgabe werden Sie einige Prozeduren im Ordner *Aufgabe 1* betrachten und implementieren. Ferner werden Sie Lemmata erstellen, widerlegen oder beweisen.

(a) Betrachten Sie die Prozedur *id*. Was macht diese Prozedur? Überprüfen Sie Ihre Vermutung indem Sie im Reiter **Interpreter** die Prozedur mit einigen Eingaben testen. Was ergibt beispielsweise $id(3)$? Danach erstellen Sie ein Lemma, welches Ihre Vermutung ausdrückt. Wählen Sie dazu im Menü **Program** den Eintrag **Insert...** und geben Sie folgendes Lemma ein, wobei Sie ??? mit Ihrer Vermutung ersetzen:

`lemma` *Vermutung* `<=` $\forall\ x : \mathbb{N}$
$id(x) = ???$

(b) Beweisen Sie Ihr Lemma, indem Sie im Menü **Program** den Eintrag **Verify** wählen. ✓`eriFun` sollte Ihr Lemma automatisch beweisen.

(c) Betrachten Sie die Prozedur $f$. Was macht diese Prozedur? Überprüfen Sie durch Testen Ihre Vermutung. Was ergibt beispielsweise $f(3, 0)$? Geben Sie der Prozedur $f$ einen sinnvollen Namen, indem Sie im Menü **Program** den Eintrag **Rename** wählen.

(d) Widerlegen Sie das Lemma „$f(x, y) = id(x) \vee f(x, y) = id(y)$", indem Sie sich ein geeignetes Gegenbeispiel überlegen und im Menü **Proof** den Eintrag **Refute** wählen. Geben Sie dann Ihr Gegenbeispiel ein, wobei Sie sowohl für $x$ als auch $y$ eine natürliche Zahl eingeben müssen.

(e) Betrachten Sie das Lemma „$f(x, y) = g(x) \vee f(x, y) = g(y)$". Dieses Lemma gilt nicht für beliebige Prozeduren $g$, insbesondere haben Sie dieses Lemma für den Fall $g = id$ widerlegt. Es gilt aber für bestimmte Prozeduren $g$. Welche Eigenschaft müsste $g$ haben? Wählen Sie die Prozedur $g$ und im Menü **Program** den Eintrag **Change Procedure...** um die Prozedur so zu implementieren, dass das Lemma stimmt. Die Prozedur $g$ darf **nicht** die Prozedur $f$ aufrufen! Benennen Sie $g$ dann sinnvoll um und beweisen Sie das Lemma. ✓`eriFun` sollte Ihr Lemma ohne weitere Interaktionen beweisen.

## Praktische Übung 1.2 (Generalisieren)

In dieser Aufgabe werden Sie im Ordner *Aufgabe 2* einige Eigenschaften über die Prozedur *reverse* beweisen. Diese arbeitet auf der in der Vorlesung vorgestellten Datenstruktur *list*[@$A$] und benutzt die Hilfsprozedur *snoc*. Ferner werden zur Spezifikation die Prozeduren *length* und # benötigt. Bevor Sie mit den Aufgaben anfangen, vergewissern Sie sich, dass Sie Datenstruktur und Prozeduren verstanden haben. Hierzu können Sie die Prozeduren mit einigen Eingaben im Reiter **Interpreter** testen.

Für die folgenden Teilaufgaben werden Sie jeweils ein Hilfslemma benötigen.

(a) Beweisen Sie das Lemma *„reverse keeps length"*.

(b) Beweisen Sie das Lemma *„reverse permutes"*.

(c) Beweisen Sie das Lemma *„reverse(reverse(k)) = k"*.

*Hinweis: Schneiden Sie den Beweisbaum für „reverse(reverse(k)) = k" direkt nach der Simplification im Schrittfall der Induktion ab, und überlegen Sie sich was für ein Hilfslemma an dieser Stelle sinnvoll wäre.*

## Praktische Übung 1.3 (Stabile Sortierverfahren)

In der Vorlesung wurde mit ✓`eriFun` nachgewiesen, dass Insertionsort ein korrektes Sortierverfahren ist: Es berechnet eine geordnete Permutation der Eingabeliste. In dieser Übung weisen wir eine weitere Eigenschaft von Insertionsort nach: Stabilität.

Um den Begriff *Stabilität* definieren zu können, betrachten wir Listen $k : \mathit{list}[\mathit{pair}[@\mathit{KEY}, @A]]$ von Paaren. Ein Paar $x : \mathit{pair}[@\mathit{KEY}, @A]$ besteht aus den Komponenten $(x)_1$ und $(x)_2$. Die erste Komponente $(x)_1$ nennen wir den *Sortierschlüssel von* $x$; nur diese Komponente soll für die Sortierreihenfolge maßgeblich sein. Ein Sortierverfahren ist *stabil*, wenn die Reihenfolge der Listeneinträge mit gleichem Sortierschlüssel unverändert bleibt.

*Beispiel:* Gegeben sei eine Liste $k$, die bereits nach der zweiten Komponente sortiert ist. Wendet man ein stabiles Sortierverfahren auf $k$ an, bekommt man eine lexikographisch sortierte Liste; das Ergebnis ist gemäß der ersten Komponente sortiert, wobei Einträge mit gleichen ersten Komponenten die ursprüngliche Reihenfolge beibehalten.

Die genannte Definition von Stabilität ist halbformal und greift auf natürlichsprachliche Begriffe wie „Reihenfolge“ zurück. Zur Formalisierung verwenden wir eine Prozedur

$$\texttt{function}\ \mathit{lookup}(z : @\mathit{KEY},\ k : \mathit{list}[\mathit{pair}[@\mathit{KEY}, @A]]) : \mathit{list}[@A]\ ,$$

die aus einer Liste $k$ die zweiten Komponenten $(x)_2$ der Paare $x = \big((x)_1 \bullet (x)_2\big)$ aus $k$ mit $(x)_1 = z$ in der Reihenfolge heraussucht, in der die Paare in $k$ vorkommen. Ein Sortierverfahren

$$\texttt{function}\ \mathit{sort}(k : \mathit{list}[\mathit{pair}[@\mathit{KEY}, @A]]) : \mathit{list}[\mathit{pair}[@\mathit{KEY}, @A]]$$

ist genau dann stabil, wenn $\mathit{lookup}(z, k) = \mathit{lookup}(z, \mathit{sort}(k))$ für alle $k : \mathit{list}[\mathit{pair}[@\mathit{KEY}, @A]]$ und alle $z : @\mathit{KEY}$ gilt.

Lesen Sie Abschnitt 2.1.3 des *$\mathcal{L}$ 1.0 Primer*, um sich mit der Syntax zur Infix- und Postfix-Notation vertraut zu machen, die zur Notation von Paaren verwendet wird. Abschnitt 1.4 in *What's New in* ✓`eriFun` beschreibt Eingabemöglichkeiten für Sonderzeichen aus dem Unicode.

(a) **(Implementierung von Insertionsort)**
Implementieren Sie die Prozedur *isort*, so dass sie eine Liste $k : \mathit{list}[\mathit{pair}[\mathbb{N}, @A]]$ von Paaren gemäß Sortierschlüssel aufsteigend (bzgl. $\leq$) und stabil sortiert. Sie können Ihre Implementierung an die aus der Vorlesung bekannte Prozedur *isort* anlehnen. Es geht hier lediglich um die Erweiterung des Listentyps von $\mathit{list}[\mathbb{N}]$ auf $\mathit{list}[\mathit{pair}[\mathbb{N}, @A]]$.

Markieren Sie dazu die Prozedur *isort*, wählen Sie im Kontextmenü *Change Procedure* und ersetzen Sie $\star$ durch Ihre Implementierung.

(b) **(Korrektheit und Stabilität von Insertionsort)**
Weisen Sie die Korrektheit und Stabilität von *isort* nach, d. h. beweisen Sie die Lemmata *isort sorts*, *isort permutes* und *isort is stable*.