Human-Computer Interaction Group

Leibniz Universität Hannover

Assignment 2

Programmieren 1 – WiSe 25/26

Prof. Dr. Michael Rohs, Falk Stock, M.Sc

Alle Assignments (bis auf das erste) müssen in Zweiergruppen bearbeitet werden. Ein Gruppenmitglied kann dabei die Lösung der Zweiergruppe gebündelt abgeben. Einzige Ausnahme ist dabei das erste Assignment, wo jedes Gruppenmitglied einzeln abgeben muss. Namen beider Gruppenmitglieder müssen sowohl in der PDF-Abgabe, als auch als Kommentar in jeglichen Quelltextabgaben genannt werden. Plagiate führen zum Ausschluss von der Veranstaltung.

Abgabe bis Donnerstag, den 30.10. um 23:59 Uhr über https://assignments.hci.uni-hannover.de/WiSe2025/Prog1. Die Abgabe muss aus einer einzelnen Zip-Datei bestehen, die den Quellcode, eine PDF für Freitextaufgaben und alle weiteren nötigen Dateien (z.B. Eingabedaten oder Makefiles) enthält. Lösen Sie Umlaute in Dateinamen auf.

Zum Bestehen der Studienleistung müssen Sie mindestens zwei von vier Punkten pro Assignment erzielen. Möchten Sie zusätzlich den Klausurbonus erreichen, müssen Sie über alle Assignments hinweg 75% der Punkte erreichen.

Aufgabe 1: Skript

Das Skript unter https://postfix.hci.uni-hannover.de/files/prog1script-postfix/ beschreibt verschiedene Vorgehensweisen bei der Lösung von Programmieraufgaben.

a) Lesen Sie Kapitel 3 (Evaluating Functions Without Side-Effects) und beantworten Sie folgende Frage in eigenen Worten: Warum können Funktionen mit Seiteneffekt nicht wie Funktionen ohne Seiteneffekte durch algebraische Umformung evaluiert werden? Geben Sie ein Beispiel für eine Funktion, die sich nicht durch algebraische Umformung evaluieren lässt.

b) Lesen Sie Kapitel 4 (Recipe for Enumerations) und beantworten Sie folgende Frage in eigenen Worten: Welche Vorteile hat die cond-Anweisung gegenüber der if-Anweisung?

c) Lesen Sie Kapitel 5 (Recipe for Intervals) und beantworten Sie folgende Frage in eigenen Worten: Welche Rolle spielen Symbole und Konstanten bei Intervallen?

d) Was war Ihnen beim Lesen der Kapitel 3 bis 5 unklar? Wenn nichts unklar war, welcher Aspekt war für Sie am interessantesten?

Aufgabe 2: Zahlenraten (1 Punkt)

Implementieren Sie ein Programm zum Zahlenraten in PostFix. Der Computer soll eine Zahl zwischen 0 und 99 würfeln, die der Spieler in möglichst wenigen Zügen herausfinden soll. Dabei gibt der Computer jeweils nur den Hinweis, ob die vom Spieler eingegebene Zahl größer oder kleiner als die gesuchte Zahl ist.

Hier die Ein- und Ausgaben für einen Beispieldurchlauf. Der Computer hat die Zahl 71 gewählt, was der Spieler natürlich nicht weiß.

```
Rate meine Zahl. Meine Zahl liegt zwischen 0 und 99.
>> 1
zu klein
>> 2
zu klein
>> 99
zu groß
>> 98
zu groß
>> 71
Richtig! Herzlichen Glückwunsch!
```

Um das Programm zu implementieren, ist folgende PostFix-Anweisung hilfreich:

```
100 rand-int            # legt eine zufällige (random) ganze Zahl im Intervall
                          [0, 100[ auf den Stack
```

a) Implementieren Sie das Programm zum Zahlenraten wie beschrieben. Achten Sie darauf, dass der Stack nach erfolgreicher Beendigung des Spiels leer ist (ohne die Verwendung von clear).

b) Spielen Sie das Spiel selbst fünf Mal. Notieren Sie jeweils wie viele Züge Sie gebraucht haben um die Zahl zu erraten. Schreiben Sie ebenfalls auf wie viele Züge Sie in Durchschnitt gebraucht haben.

c) Beschreiben Sie eine Strategie um die Zahl zu erraten, mit der möglichst wenige Züge benötigt werden.

Aufgabe 3: Formatierung von Quelltext (1 Punkt)

a) Für die Lesbarkeit von Quelltext ist die Formatierung sehr wichtig, denn Quelltext wird häufiger gelesen, als geschrieben. Gegeben sei folgender unformatierter Quelltext:

```
f: (i) { "called f" println i 0 < { i -1 * } { i 2 * } if }
fun -3 f f
```

Formatieren Sie diesen Quelltext nach folgenden Regeln:

- { ist das letzte Zeichen einer Zeile und steht nicht alleine in einer Zeile (außer bei loop)

- } ist das erste Zeichen einer Zeile, evtl. nach Leerzeichen zur Einrückung
- Zeilen in einem {...}-Block werden vier Leerzeichen tiefer eingerückt als der Block selbst
- maximal eine Anweisung pro Zeile
- komplexe Anweisungen (wie if oder fun) dürfen sich über mehrere Zeilen erstrecken

Die Regeln geben nur grobe Anhaltspunkte. Das wesentliche Kriterium ist die Maximierung der Lesbarkeit des Quelltexts. Schauen Sie sich die Formatierung der Beispiele in Tutorial und Skript Anhaltspunkte

- https://postfix.hci.uni-hannover.de/postfix-lang.html
- https://postfix.hci.uni-hannover.de/files/prog1script-postfix/

b) Erklären Sie das Verhalten der Funktion f möglichst kurz und prägnant. Welcher Wert liegt nach Ausführung des gegebenen Codes auf dem Stack?

c) Das Heron-Verfahren wurde in der iterativen Variante in PostFix implementiert (siehe: https://de.wikipedia.org/wiki/Heron-Verfahren). Die genügend gute Näherung gilt: Absolutwert(Näherung * Näherung - Zahl) < 0.01. Leider haben sich einige Fehler in die Funktion geschlichen. Formatieren Sie zuerst die Funktion, sodass sie gut lesbar ist. Finden Sie dann die Fehler und korrigieren Sie diese. Beschreiben Sie die Fehler.

```
root: (a :Num -> :Num) {x: 1 !{x = 0.5 * (x + a / x )!
a x dup * - abs 0.01 <= break if}loop}fun

"Wurzel aus 2: " print
2 root println
"Wurzel aus 4: " print
4 root println
"Wurzel aus 9: " print
9 root println
```

Aufgabe 4: Backe einen Kuchen (1 Punkt)

Gegeben sei folgendes PostFix-Programm:

BEI_200°C: 200 !

20_MINUTEN: 2000 !

100 Butter!

280 Zucker!

280 Mehl!

200 Milch!

20 Kakao!

```
6 Eier!

trennen: (AnzahlEier) {
        [AnzahlEier AnzahlEier]
} fun


vermengen: (Zutat1 Zutat2) {
        "vermenge" println
        Zutat1 Zutat2 +
        "..." println 1000 sleep
        dup print "vermengt" println
} fun


Ei-zugeben: (Menge getrenntesEi) {
        getrenntesEi 0 get str "Eigelb hinzufügen" + println
        Menge getrenntesEi 0 get 10 * +
        "..." println 1000 sleep
        getrenntesEi 1 get str "Eiweiß hinzufügen" + println
        getrenntesEi 1 get 20 * +
} fun


backen: (Menge Temperatur Zeit) {
        "vorheizen auf " Temperatur str "°C" + + println
        "vorgeheizt!" println
        "backen" println
        "..." println Zeit sleep
        Menge str "Kuchen gebacken " + println
} fun

#Rezept
Butter Zucker vermengen
Mehl vermengen
Milch vermengen
Kakao vermengen
```

Vorher	Nachher

`square: (x :Int -> :Int){`
`  x x *`
`} fun`
	`#<`
	`Diese Funktion quadriert eine Zahl.`
	`@param x Zahl die quadriert wird.`
	`@return Gibt das Quadrat der übergebenen Zahl zurück.`
	`>#`
	`square: (x :Int -> :Int){`
	`  x x *`
	`} fun`

Eier trennen

Ei-zugeben

BEI_200°C 20_MINUTEN backen

a) Welche Bezeichner verweisen auf Funktionen?

b) Was machen die einzelnen Funktionen? Geben sie an, was (welche Datentypen) die einzelnen Funktionen als Parameter auf dem Stack erwarten, was (welche Datentypen) sie als Ergebnis auf den Stack legen und was sie auf der Ausgabe ausgeben. Ergänzen Sie in der Funktionssignatur (innerhalb der .pf-Datei, keine extra Tabelle) den Typ der Parameter und ggf. den Typ des Rückgabewertes. Beispiel:

Vorher	Nachher

`square: (x){`
`  x x *`
`}fun`
	`square: (x :Int ->:Int){`
	`  x x *`
	`}fun`

c) Ergänzen Sie die Funktionsdefinitionen mit einer aussagekräftigen Parameterbeschreibung("#< ... >#"). Dazu gehören ein Purpose-Statement, die Beschreibung der einzelnen Parameter, sowie die Beschreibung des Rückgabewertes.

d) Beschreiben Sie in 1-2 Sätzen, was das Programm nachfolgend nach dem Kommentar #Rezept macht.

e) Was ist Ihre Meinung zu dem Code hinter dem Kommentar #Rezept ? Sollten Programme so geschrieben sein, dass der Programmcode annähernd wie natürliche Sprache klingt? Begründen Sie kurz.