Human-Computer Interaction Group

Leibniz Universität Hannover

Programmieren 1 – WS 2020/21

Prof. Dr. Michael Rohs, Tim Dünte, M.Sc.

Übungsblatt 2

Alle Übungen (bis auf die erste) müssen in Zweiergruppen bearbeitet werden. Beide Gruppenmitglieder müssen die Lösung der Zweiergruppe einzeln abgeben. Die Namen beider Gruppenmitglieder müssen sowohl in der PDF Abgabe als auch als Kommentar in jeglichen Quelltextabgaben genannt werden. Wenn Sie für Übungsblatt 1 noch keinen Gruppenpartner haben, geben Sie alleine ab und nutzen Sie das erste Tutorium dazu, mit Hilfe des Tutors einen Partner zu finden. Plagiate führen zum Ausschluss von der Veranstaltung.

Abgabe bis Donnerstag den 29.10. um 23:59 Uhr über https://assignments.hci.uni-hannover.de/WiSe2020/Programmieren1. Die Abgabe muss aus einer einzelnen zip-Datei bestehen, die den Quellcode, ein pdf bei Freitextaufgaben und alle weiteren nötigen Dateien (z.B. Eingabedaten oder Makefiles) enthält. Lösen Sie Umlaute in Dateinamen bitte auf.

Aufgabe 1: Skript

Unter http://hci.uni-hannover.de/files/prog1script-postfix/script.html findet sich ein Skript, das u.a. die in der Vorlesung vorgestellte Vorgehensweise bei der Lösung von Programmieraufgaben beschreibt.

a) Lesen Sie Kapitel 1 (Introduction) und beantworten Sie folgende Frage: Warum werden in dieser Vorgehensweise Beispiele für Eingaben und erwartete Ausgaben erstellt, bevor die Implementierung erfolgt?

b) Lesen Sie Kapitel 2 (Recipe for Atomic Data) und beantworten Sie folgende Frage: Warum sollte man Konstanten definieren (z.B. `WEEKLY_HOURS`), statt die entsprechenden Werte direkt ins Programm zu schreiben?

c) Was war Ihnen beim Lesen der Kapitel 1 und 2 unklar? Wenn nichts unklar war, welcher Aspekt war für Sie am interessantesten?

Aufgabe 2: Corona

Gegeben ist die Datei Corona.pf. Die gegebenen Werte und Formeln stellen nicht die Wirklichkeit dar. Es handelt sich hierbei um ein vereinfachtes Beispiel:

```
corona: (infected-people, r, epoch) {
    infected-people r epoch pow * round
} fun
```

Die Funktion corona beispielsweise berechnet die Anzahl an infizierten Personen nach einer bestimmten Anzahl an Epochen. Die vereinfachte Annahme ist, dass jemand der in Epoche x krank ist in Epoche x +1 geheilt ist. Die Anzahl der infizierten Menschen in Epoche x+1 berechnet sich

aus den Infizierten in Epoche x folgendermaßen: $Infected_{x+1} = Infected_x * r$. Wobei r den Faktor angibt, mit dem die Anzahl an Infizierten sich von Epoche zu Epoche verändert.

a) Machen Sie sich mit dem Code vertraut.

b) Geben Sie an, was (welche Datentypen) die einzelnen Funktionen als Parameter auf dem Stack erwarten und was (welche Datentypen) sie als Ergebnis auf den Stack legen. Ergänzen Sie in der Funktionssignatur den Typ der Parameter und ggf. den Typ des Rückgabewertes. Beispiel:

Vorher	Nachher

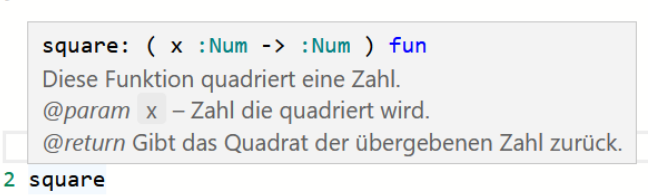
`square: (x){`
`  x x *`
`} fun`
	`square: (x :Num  -> :Num){`
	`  x x *`
	`} fun`

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

Vorher	Nachher

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

Die Postfix IDE kann dann bei Nutzung der Funktion eine automatische generierte Beschreibung als Tooltip anzeigen:



d) Warum führt der nachfolgende Aufruf von den Funktionen corona und corona2 mit gleichem r und gleicher Anzahl an Epochen zu leicht unterschiedlichen Ergebnissen:

```
100 1.1 30 corona println
100 1.1 30 corona2 30 get println
```

e) Implementieren Sie die gegebene Funktion `print-infection-array: (infected-array :Arr, infected-per-asterix :Int)`. Diese soll Arrays grafisch auf der Ausgabe

wiedergeben. Bspw. sollte beim nachfolgenden Beispielaufruf der Funktion `[1 2 3 4 5] 1 print-infection-array` folgendes ausgegeben werden:

```
* 1
** 2
*** 3
**** 4
***** 5
```

Der Parameter infected-per-asterix gibt dabei an wie viele Infizierte ein Stern (*) repräsentiert. Bspw. sollte beim nachfolgenden Beispielaufruf der Funktion `[1 2 3 4 5] 2 print-infection-array` folgendes ausgegeben werden:

```
1
* 2
* 3
** 4
** 5
```

f) Nutzen Sie die Funktion corona3 und spielen Sie mit den Parametern. Erstellen Sie 3 Aufrufe mit verschiedenen Parametern. Visualisieren Sie sich die Ergebnisse mit Ihrer in e) implementierten Funktion.

Aufgabe 3: Formatierung von Quelltext

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 die Beispiele in Tutorial und Skript an.

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

b) Erklären Sie das Verhalten der Funktion `f` möglichst kurz und prägnant.

c) Der Doktorrand Tim hat für Postfix eine Funktion geschrieben, die ihm eine Näherung für die Wurzel einer Zahl liefert. Für die Näherung nutzt Tim das Heron-Verfahren und hat es iterativ implementiert (https://de.wikipedia.org/wiki/Heron-Verfahren). Tim ist mit einer Näherung zufrieden wenn 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, die Tim gemacht hat.

```
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: Überstunden

a) Entwickeln Sie eine Funktion, `overtime`, die für eine tatsächliche Wochenarbeitszeit (in Stunden) die darin enthaltenen Überstunden berechnet. Nehmen Sie eine reguläre Wochenarbeitszeit von 36 Stunden an. Verwenden Sie nur ganze Zahlen. Verwenden Sie die im Skript unter Recipe for Atomic Data beschriebenen Schritte. Verwenden Sie die Template-Datei `overtime.pf`. Bearbeiten Sie alle mit `todo` markierten Stellen. Geben Sie in der Testfunktion ein Beispiel (Eingabe und erwartetes Resultat) für eine Arbeitszeit unter 36h, ein Beispiel für 36h und ein Beispiel größer 36h an.

b) Generalisieren Sie die in (a) erstellte Funktion so, dass die Wochenarbeitszeit als Konstante definiert wird.

Um das Programm zu implementieren, sind folgende PostFix-Anweisungen hilfreich:

2 5 test=	# test= vergleicht die beiden obersten Elemente auf dem Stack und gibt einen entsprechenden Text auf der Konsole aus
test-stats	gibt die Anzahl erfolgreicher und fehlgeschlagener Tests aus