Leibniz
Universität
Hannover

Hausübung 01

Programmieren I

23.10.2024

Norwin Bertram Wiesecke
10057811

Timon Wellhausen
10041137

Aufgabe 1:

a) Warum werden in dieser Vorgehensweise Beispiele für Eingaben und erwartete Ausgaben erstellt, bevor die Implementierung erfolgt?

Ein solches Vorgehen erlaubt es, im Vorhinein die Implementierung von Code besser zu planen. Weiterhin ist Code, bei dem die Typen von Eingabe- und Ausgabewerten fest definiert sind, weniger fehleranfällig. In PostFix können mit diesem Schritt verbunden auch parallel Testfunktionen definiert werden.

b) Warum sollte man Konstanten definieren (z.B. WEEKLY_HOURS), statt die entsprechenden Werte direkt an den benötigten Stellen ins Programm zu schreiben?

Eine Konstante zu definieren, erlaubt es einem Wert einen beschreibenden Namen zu geben, der beim Lesen des Codes die Funktionsweise klarer macht. Weiterhin kann man die Konstante später einfacher ändern, da man den Wert nur noch an der Stelle, an der die Konstante definiert wird, ändern muss, anstatt den Wert an jeder Stelle im Code zu finden und gegebenenfalls von anderen Konstanten mit gleichem Wert aber unterschiedlicher Funktion unterscheiden zu müssen. Oftmals ist es dadurch auch einfacher, die Konstante in den Code einzufügen, da man sich die Werte nicht merken muss.

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

Fachlich war beim Lesen der beiden Kapitel für uns nichts unklar. Wir hatten jedoch beide eine Art morbides Interesse für die postfix Notation der PostFix Sprache, da diese sich sämtlichen Programmiererfahrungen entgegenstellt, die wir bisher gemacht haben.

Aufgabe 2:

a) Geben Sie in eigenen Worten wieder, wofür die einzelnen Bereiche in der IDE (Editor, REPL, Input, Output, Stack & Dictionaries, Documentation) nützlich sind:

Bereich	Funktion
Editor	Text-Editor, in dem der eigentliche Code geschrieben wird.
REPL	Read-Eval-Print-Loop: Hier kann unabhängig von der Ausführung des Hauptcodes Code eingegeben werden, der Evaluiert wird und dessen Ergebnis ausgegeben wird. Werte und Variablen werden genauso auf den Stack/in das Dictionary gelegt, wie wenn Code normal ausgeführt wird. Dies kann eine große Hilfe beim Debugging sein.
INPUT	In diesem Feld kann der Nutzer Werte eingeben die durch Funktionen wie read-int, read-flt, read-char, usw. abgefragt werden. Das Feld stellt eine simple Möglichkeit dar, mit dem Programm zu interagieren.
OUTPUT	Dieses Feld funktioniert ähnlich wie eine Output-Konsole. Hier werden Werte von Funktionen wie print, println, printf, printfln ausgegeben.
Stack & Dictionaries	Zeigt die aktuell gespeicherten Werte und Variablen an, sowie deren Dateityp (:Int, :Flt, ...). Kann ebenfalls zum Verständnis und Debugging des Codes dienen.
Documentation	Ein Fenster, in dem man die Documentation der PostFix Sprache durchsuchen kann. Hierbei werden im Gegensatz zu vielen anderen Sprachen nur Details zu den verschiedenen Funktionen und Operatoren genannt.

b) Beschreiben Sie kurz, was die vier Schaltflächen zum Steuern der Programmausführung im oberen Teil der IDE bewirken.

Schaltfläche	Funktion
Run	Startet die Ausführung des Codes.
Pause	Pausiert die Ausführung des Codes, was potenziell in endless Loops nützlich sein kann oder um das Programm später an derselben Stelle weiterlaufen zu lassen (Kaffeepause).
Step Through	Führt den Code Schritt für Schritt aus, was nützlich zum Debugging ist.
Stop	Terminiert die Ausführung des Programms. Hilft als Alt-F4 Äquivalent oder wenn man den Code verändern möchte, während dieser noch ausgeführt wird.

c) Beschreiben Sie kurz, was die Menüpunkte „Add Conditional Breakpoint“, „Toggle Breakpoint“ und „Command Palette“ bewirken.

Menüpunkt	**Funktion**
Add Conditional Breakpoint	Fügt einen Punkt hinzu, an dem das Programm pausiert wird, wenn ein Boolean Wert „TRUE“ in der entsprechenden Zeile ausgegeben wird. Diese Funktion ist ebenfalls zum Debugging nützlich.
Toggle Breakpoint	Fügt einen Punkt hinzu oder entfernt ihn, an dem das Programm pausiert wird. An diesen Stellen kann man den Zustand des Programms im Stack ablesen und daraus Fehler erkennen.
Command Palette	Öffnet ein Menü mit unterschiedlichen Funktionen für die IDE. Bietet zusätzlich eine Suchfunktion zum einfacheren Navigieren.

Aufgabe 3:

a) Was passiert bei der Ausführung des folgenden Programms: 4 8 *

4:Int wird in den Stack geladen, 8:Int wird in den Stack geladen. Die beiden Obersten Werte im Stack werden multipliziert. Das Ergebnis 32:Int wird in den Stack geladen.

b) Was steht auf dem Stack, nachdem das Programm 8 3 4 2 - * ausgeführt wurde und warum?

Auf dem Stack stehen 6:Int (oben) und 8:Int (unten).

8:Int wird in den Stack geladen, 3:Int wird in den Stack geladen, 4:Int wird in den Stack geladen, 2:Int wird in den Stack geladen. Die obersten zwei Werte im Stack werden voneinander Subtrahiert (4:Int - 2:Int), das Ergebnis dieser Operation 2:Int wird in den Stack geladen. Die obersten zwei Werte im Stack (3:Int * 2:Int) werden miteinander multipliziert und das Ergebnis 6:Int wird in den Stack geladen. Die 8:Int bleibt unangerührt.

c) Wofür sind die Operatoren read-int, read-flt, print, println zuständig?

Operator	**Funktion**
read-int	Fragt einen Integer-Wert vom Nutzer über das Input Feld ab und fügt diesen zum Stack hinzu.
read-flt	Fragt einen Float-Wert vom Nutzer über das Input Feld ab und fügt diesen zum Stack hinzu.
print	Entnimmt den obersten Wert vom Stack und gibt ihn im Output Feld aus.
println	Entnimmt den obersten Wert vom Stack, gibt ihn im Output Feld aus und fügt einen Zeilenumbruch an.

d) Beschreiben Sie die Operatoren or und and. Geben Sie jeweils ein Beispiel.

Operator	**Funktion**
or	Logikoperator: Nimmt die obersten beiden Werte aus dem Stack, die Booleans sein müssen. Ist einer der beiden Werte true, ist der Output true. Der Output wird auf den Stack gelegt.
and	Logikoperator: Nimmt die obersten beiden Werte aus dem Stack, die Booleans sein müssen. Sind alle der beiden Werte true, ist der Output true. Der Output wird auf den Stack gelegt.

e) Welche der folgenden Ausdrücke sind äquivalent (hinsichtlich des Ergebnisses):
(1) 1 2 3 4 + + +
(2) 1 2 + 2 5 + +
(3) 1 2 + 3 + 4 +
(4) 3 4 + 1 2 + +
(5) 4 3 2 1 + + +
Gilt dies auch für den Operator - (Subtraktion)? Begründen Sie kurz.

Alle Funktionen haben als Ergebnis 10:Int und sind in dieser Hinsicht äquivalent. Für Subtraktion gilt dieses nicht, da dort die Reihenfolge relevant ist. Dort lauten die Ergebnisse:

2:Int,
0:Int,
-8:Int,
2:Int,
-2:Int.

f) Führen Sie das folgende Code-Fragment von Hand aus und stellen Sie den Zustand des Stacks nach jedem gelesenen und verarbeiteten Token dar: 8 2 + -1 =
Nutzen Sie als Hilfe das PostFix Execution Model aus der Vorlesung.

8:Int
8:Int, 2:Int
10:Int
10:Int, -1:Int
false:Bool

g) Was passiert, wenn Sie (nach clear) lediglich 1 / in die REPL eingeben?

Es wird folgende Fehlermeldung ausgegeben:

```
Error: Expected 2 operands but only got 1
```

Der / Operator benötigt 2 Argumente/Elemente auf dem Stack um die mathematische Operation ausführen zu können. Erhält er diese nicht, kann er die Operation nicht durchführen und gibt stattdessen eine Fehlermeldung aus.

Aufgabe 4:

a) Implementieren Sie ein Programm in PostFix, das bei Eingabe einer ganzen Zahl prüft, ob sie den Wert 100 hat. Ist das der Fall soll „Match“ ausgegeben werden. Ist der Wert kleiner 100 soll „Too Low“ ausgeben werden, ist der Wert größer 100 „Too High“. Hier soll nicht das Dictionary, sondern nur der Stack verwendet werden.

```
read-int
dup
{
    { 100 > } {"Too High" println}
    { 100 < } {"Too Low" println}
    {true} {"Match" println}
} cond
```

b) Ändern Sie das obige Programm so um, dass die Eingabe aus einer Variablen mit dem Namen „test“ gelesen wird. Weiterhin soll das Ergebnis in eine Variable „result“ geschrieben werden. Dabei soll „result“ auf 0 gesetzt werden, wenn „test = 100“ erfüllt ist, -1 wenn „test < 100“ oder 1 wenn „test > 100“. Weisen Sie der Variablen „test“ zu Beginn des Programms einen Wert zu. Prüfen Sie das Verhalten des Programms mit verschiedenen Werten für „test“.

```
read-int test!
{
    { test 100 > } {"Too High" println 1 result!}
    { test 100 < } {"Too Low" println -1 result!}
    {true} {"Match" println 0 result!}
} cond
```