Autoren Marcelo Pedernera, Jan Buehne

Aufgabe 1:

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?
Es hilft um den Algorhythmus sich mental vorzustellen und im folgenden zu implementieren. Weiterhin dienen die Beispiele und Antworten als
Testcases und sollten so gewählt werden das sie alle EdgeCases abdecken.

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?

Einerseits wiederholt man sich so nicht und ist somit im Einklang mit dem DRY-Principle und andererseits werden Funktionen, wenn man Konstanten in diesen benutzt nicht zu komplex.
Es macht kein Sinn Parameter zu übergeben die immer gleich sind.

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

Ich wusste nicht das man in PostFix auch Funktionen definieren kann. Das erleichtert im weiteren Arbeiten vieles.

Aufgabe 2:

a) Wie in Abbildung 1 zu sehen ist, hat die PostFix IDE sechs Bereiche: Editor, REPL, In, Out, Stack Dicts und Doc. Geben Sie in eigenen Worten wieder, wofür die einzelnen Bereiche in der IDE nützlich sind. Geben Sie (wenn möglich) Beispielcodefragmente (nicht aus der Vorlesung) an, um die Funktionen zu erklären. Speichern Sie Ihre Codefragmente ab und fügen Sie sie der Abgabe bei.

Editor: Hier kann eine feste Abfolge von Operatoren und Operand verknüft werden. Diese werden dann beim Ausführen des Code der Reihnfolge nach angewendet.
Es kann sowohl mit dem Stack als auch mit dem Dictionarie interargiert werden. Zudem können Objekte im Input angefordert werden und im Output ausgegeben werden.
Editor: 1 2 3 6 + + + Stack: 12
Editor: 4 x! Dicts: x 4

REPL: Da PostFix Zeile für Zeile interpretiert wird, kann mit dem REPL, während der Code ausgeführt wird, einzelnene Operationen einschieben.
z.B. REPL: 4 4 +
STACK: 8

IN: Hier können Objekte eingegeben werden, wenn diese mit dem read Operator abgefragt werden.
z.B.: Editor: readint print
IN: 7
OUT: 7

OUT: Hier werden Objekte ausgegeben, welche durch den print Operator ausgegeben werden.
z.B.: Editor: 5 2 + print +
OUT: 7

Stack Dicts: Jeder neue Eintrag wird auf den Stack gelegt. Es kann das oberste Objekte vom Stack entfernt/verwendet werden oder ein neustes oberstes Objekte darauf gelegt werden.
Im Dictionarie werden bestimmten Variablen Objekten zugeordnet. Diese können dann jederzeit abgerufen oder ersetzt werden.

Doc: In der Documentation sind alle Operatoren und deren Anwendung beschrieben. Es ist wie eine Bedinungsanleitung für die einzelnen Bauteile des Codes.
z.B. Doc: for
dann spuckt mir das aus:
lower :Int, upper :Int, body :ExeArr
Execute the given executable array for every even number between lower (inclusive) and upper (exclusive). Before every iteration, the current value is pushed on the stack.

Params:
lower:Int Start of the range (inclusive)
upper :Int End of the range (exclusive)
body :ExeArr Executable array to run for number in the range
Returns: nothing
Example:
1 10 { println } for # prints the numbers from 1 to 9

b) Die erste Schaltfläche führt alle Anweisungen aus dem Editor in Reihnfolge aus.
Die zweite Schaltfläche pausiert das Programm in dem aktuellen Schritt.
Die dritte Schaltfläche führt die Anweisungen aus dem Editor Schritt für Schritt aus.
Die vierte Schaltfläche stoppt das Programm und geht wieder zum Start.

c) "Add conditional Breakpoint"
Added einen Breakpoint der an eine Condition geküpft ist (Bei True ist der Breakpoint active)

"Toggle Breakpoint"
Fügt einen Breakpoint hinzu oder entfernt ihn
Breakpoint = Pausierpunkt

"Command Palette"
Ruft eine ähnliche Command Pallete zu VSCode auf, mit ShortCuts die die IDE anbietet

Aufgabe 3: a) Es wird erst 4 dann 8 auf den Stack gelegt und dann 4*8=32 gerechnet und die 32 danach auf den Stack gelegt.

b) Wenn der Stack aus 8 3 4 2 - * besteht wird erst 4-2=2 und dann 2*3=6 gerechnet. Damit ist der Stack am Ende 8 6.

c) Read-Int erlaubt den Input eines Integers (Ganzzahl), Read-flt erlaubt den Input eines Floats (Fließkommazahl), Print nimmt das oberste Element des Stacks und gibt es im OutputFeld aus
Println macht das gleiche, fügt aber einen Zeilenumbruch (\n) an das Ende der Zeile an.

d) Der Operator or vergleicht zwei Wahrheitswehrte und gibt true aus solange einer der beiden Wahrheitswehrte true ist. Ansonsten wird false ausgegeben. Entspricht dem logischen oder aus der Aussagenlogik.
Der Operator and vergleicht zwei Wahrheitswehrte und gibt true aus solange beide Wahrheitswehrte true sind. Ansonsten wird false ausgegeben. Entspricht dem logischen und aus der Aussagenlogik.

e) (1) 1 2 3 4 + + + bedeutet 3+(4+(2+1))=10
(2) 1 2 + 2 5 + + bedeutet (1+2)+(2+5)=10
(3) 1 2 + 3 + 4 + bedeutet ((1+2)+3)+4=10
(4) 3 4 + 1 2 + + bedeutet (3+4)+(1+2)=10
(5) 4 3 2 1 + + + bedeutet ((1+2)+3)+4=10
Damit sind alle Ausdrücke äquivalent.
(1) 1 2 3 4 - - - bedeutet 3-(4-(2-1))=-2
(2) 1 2 - 2 5 - - bedeutet (1-2)-(2-5)=2

(3) 1 2 - 3 - 4 - bedeutet ((1-2)-3)-4=-8
(4) 3 4 - 1 2 - - bedeutet (3-4)-(1-2)=0
(5) 4 3 2 1 - - - bedeutet ((1-2)-3)-4=2
Damit sind die Ausdrücke (2) und (5) hinsichtlich des Ergebnisses äquivalent.
Es verhält sich hier nicht genau wie bei der Addition, da für die Subtraktion das Kommutativgesetz nicht gilt

f) Zuerst wird 8 auf den Stack gelegt. Danach wird 2 auf den Stack gelegt. Dann wird 8+2=10 gerechnet.
Dann wird -1 auf den Stack gelegt und dann wird überprüft ob 10=-1. Das Ergebnis davon ist false und das bleibt dann auf dem Stack.

g) Es gibt eine Fehlermeldung aus, da eine Division nur von zwei Operanden ausgeführt werden kann und bei 1 / nach einem clear nur 1 Operand auf dem Stack liegt.

Aufgabe 4: a) read-int dup 100 = {"Match" print} {100 > {"Too High" print} {"Too low" print} if} if
b) Hier wird durch das rand-int keyword und durch den loop n-1 beliebige Werte zwischen 0 und x generiert und die Ergebnisse werden sauber formatiert zur Kontrolle im Outputfield ausgegeben
"""
Generate n - 1 random testcases between 0 and x
200 x!
20 n!

1 n {
    x rand-int test!
    test dup 100 = {0 result!} {100 > {1 result!} {-1 result!} if} if
    "Inputed Number:" print test print " Result:" print result println
} for
"""

PostFix Code ist auch nochmal seperat in einer einzelnen Datei: