Discord: timostucki

Meine Aufgaben Ratings:

- Best Case: Ihr macht alle Aufgaben (, wenn die Zeit reicht)
- Falls nicht: Ich mache Ratings zur Wichtigkeit der einzelnen Aufgaben

- (Keine offizielle Empfehlung, meine subjektive Meinung auf Basis meiner eigenen Erfahrung als Student in diesem Kurs)
- ! Sagen nichts über die Schwierigkeit der Aufgaben aus !

Wichtig für die Prüfung:
(Prüfungs- oder
prüfungsähnliche Aufgaben)

Wichtig für euer Verständnis:
(Aber nicht in Prüfungsform)

Wichtig für tiefes Verständnis/
Zusatzaufgaben:
(Bspw. viele vom gleichen Typ)

Aufgabe 1: Loop-Invariante

Gegeben ist die Methode `findLargestSmaller(int[] al, int value)`, die in einem *sortierten nicht-leeren* Array von `int`-Werten den grössten Wert findet, der strikt kleiner als `value` ist. `value` muss strikt grösser als `al[0]` sein.

Was ist die Invariante für den Loop in der Methode `findLargestSmaller`? Wenn die Elemente `a[0]` .... `a[K]` des Arrays a sortiert sind, dann können Sie das mit `Sorted(a, 0, K)` abkürzen.

```
static int findLargestSmaller(int[] al, int value) {
    // Precondition: Sorted(al, 0, al.length-1) && al.length >= 1 && value > al[0]
    int candidate  = al[0];
    int next = 1;

    // Loop Invariante:
    while (next < al.length && value > al[next] ) {
        candidate = al[next];
        next++;
    }

    // Postcondition: Sorted(al, 0, al.length-1) && al.length >= 1 &&
        ((next < al.length && value <= al[next] && candidate == al[next-1]) ||
        (next == al.length && candidate == al[al.length-1] && value > candidate))

    return candidate;
}
```

Aufgabe 2: Maps

1. Implementieren Sie eine Methode `U11Map.arrayToMap(String[] A)`, die einen Array `A` von Strings als Parameter akzeptiert und eine Map `M` von String zu Integer zurückgibt. Jeder String, der in `A` auftritt, soll in `M` auf die Zahl 0 abgebildet werden. Sie können davon ausgehen, dass `A` nicht `null` ist und dass kein String der empty (leere) String ist.

   Zum Beispiel gibt `arrayToMap` für den Array `[one, two, three, one]` die Map `{one=0, two=0, three=0}` zurück.

2. Implementieren Sie eine Methode `U11Map.arrayToMapOne(String[] A)`, die einen Array `A` von Strings als Parameter akzeptiert und eine Map `M` von String zu Integer zurückgibt. Jeder String, der in `A` auftritt, soll in `M`, falls der String nur einmal vorkommt, auf die Zahl 0 abgebildet werden und, falls der String mehrfach vorkommt, auf die Zahl 1 abgebildet werden. Sie können davon ausgehen, dass `A` nicht `null` ist und dass kein String der empty (leere) String ist.

   Zum Beispiel gibt `arrayToMapOne` für `[one, two, three, one]` die Map `{one=1, two=0, three=0}` zurück und für `[one, two, three, one, one, four, two]` die Map `{four=0, one=1, two=1, three=0}`.

Aufgabe 3: Generische Listen

In dieser Aufgabe implementieren Sie eine generische verkettete Liste. Anhang A zeigt eine generische Version eines Interfaces für Listen. Vervollständigen Sie die Klasse `MyListImpl`, sodass die Klasse das `MyList` Interface implementiert. Dem Interface wurden zwei neue Methoden hinzugefügt. Die Methode `MyListNode getNode(int index)` gibt den Knoten zurück, welcher den Wert der Liste an Position `index` speichert. `MyListeNode` ist selber ein Interface (siehe Anhang B) mit Methoden, welche jeweils den gespeicherten Wert des Knoten, den nächsten Knoten, und ob es einen nächsten Knoten gibt zurückgeben. Damit überprüfen wir, dass `MyListImpl` eine verkettete Liste implementiert. Die Methode `Iterator<T> iterator()` gibt einen Iterator für die Datenstruktur zurück. Implementieren Sie einen neuen Iterator, das heisst eine Klasse, welche das `Iterator` Interface implementiert, und geben Sie nicht den Iterator einer anderen Datenstruktur zurück (zum Beispiel den Iterator einer ArrayList). Dies können wir in den Tests der Korrektur testen. Die `void remove()` Methode vom Iterator wird nicht benötigt. Die Methode `void addAll(MyList<T> other)` sollte eine konstante Laufzeit haben. Ein paar Tests finden Sie in `MyListTest`.

Aufgabe 4: Notenauswertung

Die Klasse `Service` stellt verschiedene Analysen für Prüfungsergebnisse von S Studierenden zur Verfügung. Die Liste von Ergebnissen besteht aus S Einträgen, also jeweils ein Eintrag pro Student/in. Jeder Eintrag besteht aus einer Zeile und enthält (in dieser Reihenfolge):

1. die Immatrikulationsnummer des Studierenden (ein identifizierender positiver `int`-Wert)
2. drei Noten (drei reelle Zahlen im Bereich von 1.0 bis 6.0, getrennt durch Leerzeichen)

Die drei Noten gehören zu den Fächern *Fach 1*, *Fach 2* und *Fach 3*. Zusätzliche Leerzeilen und -zeichen sollen ignoriert werden. Eine Beispiel für eine Liste für 3 Studierende ist:

```
111111004  5.0  5.0  6.0
111111005  3.75 3.0  4.0
111111006  4.5  2.25 4.0
```

Ihre Aufgabe ist es nun, die `Service`-Klasse und ihre Analysen zu implementieren. Die `Service`-Klasse hat einen Konstruktor, welcher alle Prüfungsergebnisse aus einem `Scanner` auslesen und damit das `Service`-Objekt initialisieren soll. Das Objekt soll so initialisiert werden, dass die vorgegebenen Methoden ihre Analysen durchführen können. Sie dürfen dabei Attribute und zusätzliche Methoden frei bestimmen.

Aufgabe 4: Notenauswertung

a) Implementieren Sie nun die Methode `critical()`, welche die zwei Argumente `bound1` und `bound2` erwartet. Die Methode sucht alle "kritischen" Fälle und gibt eine Liste dieser Studierenden zurück. Ein Student darf maximal einmal in der Liste vorkommen. Die zurückgegebene Liste besteht aus den Immatrikulationsnummern dieser Studierenden (in beliebiger Reihenfolge).

   Ein/e Student/in gilt als kritisch, wenn die Note in *Fach 1* $\leq$ `bound1` und die Summe der Noten für *Fach 2* und *Fach 3* kleiner als `bound2` ist.

   Für das obige Beispiel gäbe `critical(4, 8)` eine Liste mit dem Element `111111005` zurück.

b) Implementieren Sie nun die Methode `top()`, welche die Studierenden mit den besten Ergebnissen zurückgeben soll. Der Parameter `limit` bestimmt die maximale Anzahl der zurückzugebenden Studierenden. Falls weniger Ergebnisse als `limit` existieren, sollen einfach alle gefundenen zurückgegeben werden.

   Der Rückgabewert der Methode ist wieder eine Liste der Immatrikulationsnummern. Ein Student darf maximal einmal in der Liste vorkommen. Diese Liste soll absteigend nach der Leistung sortiert sein (der/die Student/in mit dem besten Ergebnis zuerst). Dabei gilt, dass ein Ergebnis *A* besser ist als ein Ergebnis *B*, wenn die Summe aller Noten von *A* grösser ist als die Summe der Noten von *B*. Sind die Summen gleich, sind die Ergebnisse gleich gut (und die Reihenfolge in der Liste somit egal).

   Für das obige Beispiel gäbe `top(2)` entweder die Liste `[111111004, 111111006]` oder die Liste `[111111004, 111111005]` zurück (beide wären richtig).

Aufgabe 5: Interpreter

In der letzten Übung implementierten Sie einen Evaluator für mathematische Ausdrücke. In dieser Aufgabe erweitern Sie ihn so, dass er statt einzelnen Ausdrücken einfache Programme mit mehreren Anweisungen auswertet. Das nennt man auch *interpretieren* und ein solches Programm entsprechend *Interpreter*.

```
digit  ← 0 | 1 | ... | 9
char   ← A | B | ... | Z | a | b | ... | z
num    ← digit { digit } [ . digit { digit } ]
var    ← char { char }
func   ← char { char } (
op     ← + | - | * | / | ^
atom   ← num | var
term   ← ( expr ) | func expr ) | atom
expr   ← term [ op term ]
stmt   ← var = expr ;
prog   ← { stmt }
```

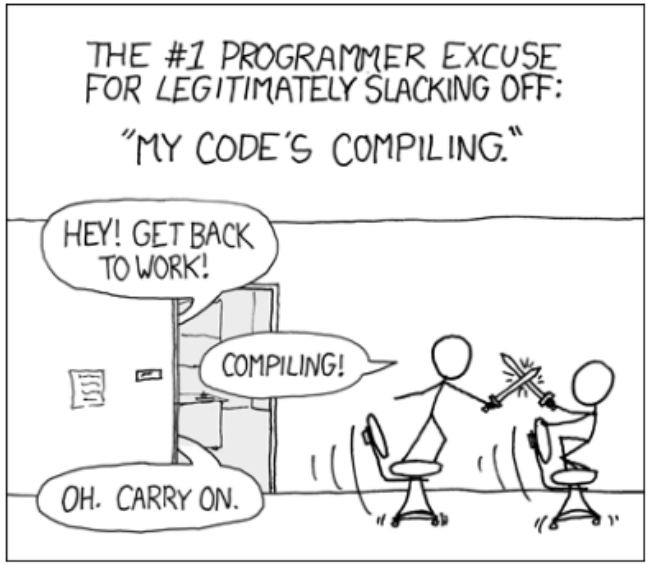
Abbildung 1: **EBNF-Beschreibung** von *prog*

```
alpha = i * ((2*PI) * (1 / 6.05));
size = (0.25 * cos(t/2)) + 0.75;

x = cos(alpha + (0.3 * t)) * size;
y = sin((1.5 * alpha) + t) * size;

r = (cos(alpha + (2 * t)) + 1) / 2;
g = (sin(alpha + (2 * t)) + 1) / 2;
b = (cos(alpha +  (PI/2)) + 1) / 2;
```

Abbildung 2: Beispiel-Programm

Aufgabe 6: Compiler

Das Interpretieren von Quellcode ist ineffizient und langsam. Deshalb werden Java-Programme auch zuerst *kompiliert*, bevor sie ausgeführt werden. Kompilieren heisst, den Quellcode in eine Form zu übersetzen, die vom Computer direkt(er) ausgeführt werden kann. In dieser Übung schreiben Sie einen einfachen Compiler, der den Quellcode von Programmen von Aufgabe 5 in eine Serie von Instruktionen übersetzt, die effizient ausgeführt werden können.



xkcd: Compiling by Randall Munroe (CC BY-NC 2.5)

Die Programmiersprache in Aufgabe 5 hat eine rekursive Struktur: Ausdrücke können mehrere Ausdrücke enthalten, welche wiederum mehrere Ausdrücke enthalten können, usw. Um eine solche Struktur in eine lineare Folge von Instruktionen umzuwandeln, verwenden wir einen *Operanden-Stack*. Dies ist ein `Stack` (wie Sie Ihn in der Vorlesung gesehen haben), der Zwischenresultate von Berechnungen speichert. Instruktionen können Werte auf den Stack "pushen" oder Werte ab dem Stack "poppen" und verwenden. Es gibt folgende Arten von Instruktionen:

Aufgabe 6: Compiler

Programmteil	Instruktionen
`b`	`LOAD b`
`1`	`CONST 1`
`b + 1`	`LOAD b CONST 1 OP +`
`(b + 1)`	`LOAD b CONST 1 OP +`
`2`	`CONST 2`
`c`	`LOAD c`
`2 * c`	`CONST 2 LOAD c OP *`
`sin(2 * c)`	`CONST 2 LOAD c OP * FUNC sin`
`(b + 1) / sin(2 * c)`	`LOAD b CONST 1 OP + CONST 2 LOAD c OP * FUNC sin OP /`
`a = (b + 1) / sin(2 * c);`	`LOAD b CONST 1 OP + CONST 2 LOAD c OP * FUNC sin OP / STORE a`

Tabelle 1: Kompilieren der Anweisung `a = (b + 1) / sin(2 * c);`

`CONST` c	Pusht den konstanten Wert c auf den Stack
`LOAD` v	Lädt den Wert der Variable v und pusht ihn auf den Stack
`STORE` v	Poppt einen Wert vom Stack und speichert ihn in der Variable v
`OP` $\oplus$	Poppt zwei Werte r und l vom Stack (zuerst r, dann l) berechnet $l \oplus r$ und pusht das Resultat zurück auf den Stack
`FUNC` f	Poppt einen Wert x vom Stack, berechnet $f(x)$ und pusht das Resultat zurück auf den Stack

Unten sehen Sie ein kleines Programm, das aus solchen Instruktionen besteht. Es lädt zuerst den Wert der Variable `x` und dann einen konstanten Wert 2 auf den Stack. Die nächste Instruktion holt sich die beiden Werte vom Stack, multipliziert sie und pusht das Resultat zurück. Die letzte Instruktion schliesslich holt diesen Wert vom Stack und speichert ihn zurück in die Variable `x`:

```
LOAD x
CONST 2
OP *
STORE x
```