

Artemis 6.4.0

Course Overview

ge64baw

Courses > Praktikum: Grundlagen der Programmierung WS22/23 > Exercises > W13H01 - Von Teilern und Vielfachen

W13H01 - Von Teilern und Vielfachen

Hausaufgabe

Easy

Submission due: 6 months ago

Points: 9 of 8 (Bonus: 2) Assessment: automatic ?

Complaint due: 6 months ago

 112.5% (6 months ago) GRADED

Recent results:



Show all results

Tasks:

Von Teilern und Vielfachen

Die Forschuine an der PUM haben aus Versehen ihren MiniJava-Compiler kaputt gemacht! Ohne diesen sind sie allerdings ganz schön aufgeschmissen, denn viele Prozesse an der PUM sind auf den Compiler angewiesen. Daher brauchen sie mal wieder Deine Hilfe um den Tagesablauf aufrecht zu erhalten und die Grundfunktionalität wiederherzustellen.

Update 26.1 19:30

- Im Template für die Bonusaufgabe soll es `FJUMP label` (ohne `:`) und `NEG` in Zeile 20 heißen. Das korrigierte Template findet ihr [hier](#). Für alle die die Aufgabe nach diesem Zeitpunkt begonnen haben ändert sich nichts.

Kompilieren von MiniJava-Programmen

Eine sehr häufig verwendete mathematische Berechnung ist das kleinste gemeinsame Vielfache - auf englisch auch das "least common multiple", kurz `lcm`. Der Algorithmus berechnet für zwei Zahlen `a` und `b` die kleinste Zahl, welche von beiden ohne Rest teilbar ist. Für `a = 7` und `b = 2` wäre `lcm(7, 2) = 14`, für `a = 6` und `b = 10` wäre `lcm(6, 10) = 30`. Da man nicht durch null teilen kann, ist das Verfahren nur für `a, b ≠ 0` definiert.

Euch ist nun ein Programm in MiniJava gegeben, welches zuerst zwei Zahlen `a` und `b` mit Hilfe der Instruktion `READ` einliest und dann deren kleinstes gemeinsames Vielfaches berechnet, welches dann wiederum am Ende mit Hilfe von `WRITE` ausgegeben wird. Eure Aufgabe ist es nun, dieses Programm **genau nach den Regeln aus der Vorlesung (bzw. Zentralübung)** in MiniJava-Bytecode zu übersetzen. Dabei ist wichtig, dass Ihr wirklich genau die Regeln der Vorlesung (bzw. Zentralübung) befolgt. Interpretationen wie z.B. `a = a * -1` statt `a = -a` zu übersetzen werden (wenn auch semantisch equivalent) als falsch gewertet!

```
int a, b, r;
a = read();
b = read();
if (a <= 0) {
    a = -a;
}
if (b <= 0) {
    b = -b;
}
r = a * b;
while (a != b) {
    if (b < a) {
        a = a - b;
    } else {
        b = b - a;
    }
}

r = r / a;
write(r);
```

Abgabe und Tests

- Euer Programm müsst Ihr in die Datei `lcm.jvm` schreiben, welche im Template bereits vorgegeben ist. Änderungen an Dateiname oder Speicherort können dazu führen, dass die Aufgabe nicht bewertet werden kann.
- Kommentare könnt ihr wie bereits aus Java bekannt mit `//` schreiben. Alles, was in einer Zeile nach dem `//` folgt, wird von unserem Interpreter ignoriert.

- Jede Instruktion sollte eine eigene Zeile bekommen. Nach einem Label darf noch eine weitere Instruktion folgen. Auf Groß- und Kleinschreibung muss nicht geachtet werden.
- Beachtet auch die Folge der Instruktionen. Dabei gilt, dass das oberste Element auf dem Stack rechts vom Operator steht und das zweit oberste links. $x = x + 1$ übersetzt sich daher genau zu:

```
LOAD 0 // das vorletzte Element ist links von dem +
CONST 1 // und das letzte Element rechts von dem +
ADD
STORE 0
```

(unter der Annahme natürlich, dass Variable x in Speicherzelle 0 gehalten wird.)

- Die Tests für diese Aufgabe sind (obwohl es eine H01 ist) in public und hidden unterteilt.
- Ihr könnt einmal zwei Punkte vor der Deadline sicher sammeln, wenn Euer Programm für zwei eingegebene a und b das richtige Ergebnis $lcm(a, b)$ berechnet.
- Nach der Deadline wird Eure Abgabe automatisch korrigiert und es werden für richtige Codeblöcke die restlichen sechs Punkte vergeben.

✓ Der ByteCode in lcm.jvm kann interpretiert werden [1 of 1 tests passing](#)

✓ Der ByteCode in lcm.jvm berechnet für gegebene Eingaben die korrekten Ausgaben [1 of 1 tests passing](#)

✗ Der ByteCode in lcm.jvm ist syntaktisch korrekt (nach der Deadline) [5 of 7 tests passing](#)

Bonus: Dekompilieren von ByteCode

Des Weiteren haben die Forscher auf ihrem Server noch einen Fetzen Bytecode gefunden. Dieser scheint merkwürdige Berechnungen mit dem kleinsten gemeinsamen Vielfachen zu machen, die sie gern verstehen würden. Daher bitten sie Dich datenforensisch aktiv zu werden und das MiniJava-Programm wieder herzustellen, welches der Ursprung dieses Bytecodes war.

```
ALLOC 2

READ
STORE 0

READ
STORE 1

LOAD 0
LOAD 1
MUL
STORE 0

LOAD 0
CONST 0
LESS
FJUMP label

LOAD 0
NEG
STORE 0

label:

LOAD 0
LOAD 1
CALL lcm
STORE 1

LOAD 0
LOAD 1
DIV
STORE 1

LOAD 1
WRITE
HALT
```

Aber Achtung, in dem Bytecode kommt eine Instruktion vor, die bisher noch nicht behandelt wurde. Die Instruktion **CALL fun** wird verwendet, um eine Methode mit dem Namen **fun** auszuführen und funktioniert dabei so:

1. Wenn **fun** n Argumente entgegen nimmt, dann werden n Elemente vom Stack entfernt, wobei das oberste Element das letzte Argument der Methode **fun** ist, das zweite Element das vorletzte Argument und so weiter.
2. Dann wird die Methode **fun** ausgeführt.
3. Der Rückgabewert der Methode wird oben auf den Stack gelegt.

Die Methode `lcm` ist dabei die Gleiche wie aus der ersten Teilaufgabe, nur dass die Parameter nicht eingelesen / geschrieben werden sondern Argumente / Rückgabe sind. Die Methode hat daher die Signatur `int lcm(int a, int b)`.

Beispiel für `CALL`

```
int a;
a = Math.pow(3, 2);
```

Würde man wie folgt übersetzen:

```
ALLOC 1
CONST 3
CONST 2
CALL Math.pow
STORE 0
```

Abgabe und Tests

- Den Bytecode findest Du noch einmal in der Datei `fragment.jvm`.
- Das dekompierte Programm soll in den Körper der Methode `fragment()` in der Klasse `Fragment` geschrieben werden. Der Methodenkopf sowie alle weiteren Methoden in der Klasse dürfen **nicht** verändert werden.
- In der Klasse `Fragment` sind auch schon Methoden für `READ` und `WRITE` gegeben, sowie die Methode, welche von `CALL` aufgerufen wird vorgegeben, diese **müssen** verwendet werden.
- Die Tests geben einen Punkt auf semantische Korrektheit (daher, dass Programm funktioniert genau wie `fragment.jvm`) und einen Punkt auf syntaktische Korrektheit (daher, der Code ist **genau** der Gleich welcher nach den Regeln der Vorlesung - und der neuen Definiton von `CALL fun` - zu `fragment.jvm` kompiliert wurde).

- ✔ Die Methode `fragment` kann bewertet werden [1 of 1 tests passing](#)
- ✔ Dekompilieren von Bytecode [2 of 2 tests passing](#)

FAQ

if ohne else: Wenn ein `if`-Statement keinen `else`-Teil hat, dann muss dies nach den Regeln aus der Zentralübung übersetzt werden.
JUMP zu Zeile: Bei einem `JUMP` Befehl kein Label sondern eine Zeile anzugeben wird momentan nicht unterstützt.

Lösungsvorschlag

Tests

Lokal ausführen: Damit die Tests lokal funktionieren müsst ihr in der Klasse `BehaviorTest` die Methode `readFile` anpassen. Dort steht auch wie genau

Exercise details

Release date:	Jan 26, 2023 18:30
Start date:	Jan 26, 2023 18:30
Submission due:	Feb 12, 2023 18:00
Complaint due:	Feb 19, 2023 18:00

Every student is allowed to complain once per exercise. In total 1000 complaints are possible in this course. You still have **998** complaints left. ⓘ