



Artemis 6.4.0

 Course Overview



 ge64baw ▾


Courses > [Praktikum: Grundlagen der Programmierung WS22/23](#) > Exercises > [W13H03 - Pingu JVM](#)

 **W13H03 - Pingu JVM**

Hausaufgabe

Hard

Submission due: **6 months ago**

Points: 6 of 6 Assessment: automatic 

Complaint due: **6 months ago**

 [100% \(6 months ago\)](#)

GRADED

Recent results:



Show all results ▾

Tasks:

W13H03 - Pingu JVM

Die Studentuine sind begeistert von dem Bytecode-Simulator aus den Übungen, den du ihnen gezeigt hast. Da man in der Antarktis aber einen anderen Bytecode-Dialekt spricht und der Befehlssatz in unserem Simulator gehardcoded ist, wollen sie nun einen eigenen Simulator für ihre Bytecode-Variante schreiben. Dabei sind sie etwas übermütig geworden und haben gleich eine 5-Schritte-Pipeline implementiert, mithilfe derer ein Bytecode-Programm aus einer Datei eingelesen, in logische Blöcke zerlegt, verifiziert, optimiert und letztlich simuliert werden soll. Nun ist das Ganze aber ein bisschen außer Hand geraten und deine Kommiliton:innen von der PUM brauchen Deine Hilfe, damit ihr Projekt nicht auseinanderfällt. Kannst Du sie unterstützen?

Allgemeines

Jeder Teil soll die schon im Template vorgegebene und genutzte Struktur nutzen. Um zu verhindern, dass Fehler in früheren Schritten das Testen späterer Schritte unmöglich machen, darf sie nicht verändert werden um z.B. weitere Parameter hinzuzufügen. Bei der Implementierung von jedem Schritt kann davon ausgegangen werden, dass alle vorherigen Schritte korrekt implementiert sind und entsprechend die darin überprüften Regeln von der Eingabe auch eingehalten werden.

Der Befehlssatz

Wir beschäftigen uns in dieser Aufgabe mit einer Variante des aus der Vorlesung und den Zentralübungen bekannten MiniJava-Bytecodes, der außer den uns bekannten Befehlen noch die drei Befehle **POP**, **DUP** und **SWAP** einführt. **POP** entfernt dabei das oberste Element auf dem Stack, ohne damit etwas zu machen, **DUP** dupliziert das oberste Element (also legt eine Kopie des bisher obersten Elementes noch oben auf den Stack drauf) und **SWAP** vertauscht die obersten beiden Elemente des Stacks.

Zudem unterscheiden wir bei dieser JVM auch zwischen dem Variablenteil des Stack (im Folgenden als "lokale Variablen" bezeichnet), der am Anfang durch **ALLOC** angelegt wird und dem Rest des Stacks, auf dem die eigentlichen Berechnungen stattfinden. Die Befehle **LOAD i** und **STORE i** können dann nur aus dem Variablenteil laden bzw. auf ihn speichern. Andere Befehle, wie **ADD**, **AND**, **LESS** etc., arbeiten ausschließlich auf dem oberen, restlichen Teil des Stacks. D.h. folgende beiden Programme sind in dem in dieser Aufgabe betrachteten Dialekt nicht zulässig:

▼ Unzulässige Bytecode-Programme

READ
LOAD 0
HALT

CONST 1337
ALLOC 2
HALT

ALLOC 2
READ
STORE 0
READ
STORE 1
ADD
WRITE
HALT

Hier nochmal eine Übersicht alle in diesem Bytecode-Dialekt zulässigen Befehle.

Typ	Befehl
int-Operatoren:	NEG, ADD, SUB, MUL, DIV, MOD
boolean-Operatoren:	NOT, AND, OR
Vergleichs-Operatoren:	LESS, LEQ, EQ, NEQ
Laden von Konstanten:	CONST i, TRUE, FALSE
Speicher-Operationen:	LOAD i, STORE i
Sprung-Befehle:	JUMP i, FJUMP i
IO-Befehle:	READ, WRITE
Reservierung von Speicher:	ALLOC i
Stack-Befehle:	POP, DUP, SWAP
Beendung des Programms:	HALT

Basic Blocks

Wir stellen uns ein Bytecode-Programm als in einzelne Blöcke unterteilt vor. Dabei ist ein Block jeweils die größte Menge an aufeinanderfolgenden Instruktionen, sodass

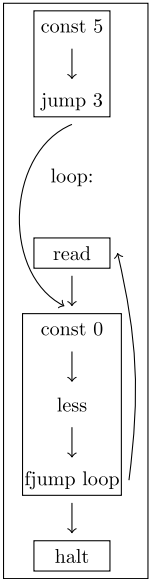
- 1. ein **JUMP**, **FJUMP** oder **HALT** nur als letzte Instruktion eines Blocks vorkommt und
- 2. durch Sprünge (oder den Anfang des Programms insgesamt) nur zum Anfang eines Blocks gesprungen werden kann.

Um ein gegebenes Programm in Blöcke zu unterteilen, beginnen wir also ganz vorne und gehen das Programm Instruktion für Instruktion durch. Jedes Mal, wenn wir an eine Zeile kommen (ob durch Marker oder Zeilennummer gekennzeichnet, ist egal), an die irgendwo im Programm aus gesprungen wird, beginnen wir *vor* dieser Zeile einen neuen Block. Jedes Mal, wenn wir an einen Sprung-Befehl (oder **HALT**) kommen, beginnen wir *hinter* der Zeile mit dem Befehl einen neuen Block.

▼ Hier ein Beispiel dazu:
Das Programm

```
const 5
jump 3
loop:
read
const 0
less
fjump loop
halt
```

(nur Zeilen mit Instruktionen haben Zeilennummern, Zeile 3 enthält also die Instruktion **const 0**) wird folgendermaßen in Blöcke aufgeteilt:



Die Pipeline

Das fertige Programm soll eine Datei mit Bytecode einlesen und diesen letztlich ausführen können. Dazwischen gibt es noch einige Schritte, die erledigt werden müssen. In diesem Abschnitt wird die Pipeline, durch die der eingelesene Bytecode-Text geschickt wird, im Ganzen beschrieben, in den folgenden Abschnitten erfährst Du dann, was Du davon eigentlich erledigen sollst.

Die Pipeline besteht aus fünf Schritten:

I. Parsing: In diesem Schritt wird die übergebene Datei eingelesen und es werden die Instruktionen extrahiert. Dabei wird ein mögliches **ALLOC** am Anfang weggeschnitten und der restlichen Pipeline wird nur übergeben, wie viele "lokale Variablen es gibt", soll heißen wie viel Speicher mit diesem **ALLOC** alloziert wurde. Eingabe ist also Pfad zu einer Datei (hoffentlich mit Bytecode als Inhalt), Ausgabe ist ein Array mit Instruktionen und die Anzahl an lokalen Variablen.

II. Splitting: In diesem Schritt werden die einzelnen Instruktionen in Blöcke zusammengefasst, wie in vorherigem Abschnitt beschrieben. Eingabe ist also die Ausgabe des vorherigen Pipeline-Schrittes, Ausgabe ist eine Liste mit allen Blöcken (und wieder die Anzahl lokaler Variablen).

III. Verifying: In diesem Schritt wird sichergestellt, dass das Bytecode-Programm valide ist und ausgeführt werden kann. Dabei werden Dinge geprüft, wie ob wenn ein **ADD**-Befehl ausgeführt wird, sicher auch zwei Integer ganz oben auf dem Stack liegen und nicht etwa ein Boolean darunter ist oder gar weniger als zwei Elemente auf dem Stack liegen. Dazu wird das Bytecode-Programm Block für Block simuliert, allerdings nicht mit konkreten Werten, sondern bloß mit den auf dem Stack liegenden Typen. Simulation eines **LESS** Befehls verlangt hier z.B. zweimal Typ **INT** oben auf dem Stack (sonst wird ein Error geworfen), löscht beide und legt einmal den Typ **BOOL** oben ab. Bei Verzweigungen (**FJUMP**) werden beide Zweige des Programms simuliert. Eingabe ist hier die Ausgabe von Schritt II, Ausgabe sind die leicht modifizierten (siehe unten) Blöcke, wieder die Anzahl an lokalen Variablen und zuletzt noch die höchste Höhe, die der Stack in der Simulation erreicht hat. Wichtig bei diesem Schritt ist aber vor allem, dass er überhaupt ohne Exceptions durchläuft. Das garantiert dann, dass das Bytecode-Programm auch wirklich ausführbar ist.

IV. Optimizing: In diesem Schritt können wir nun den Verifizierten Bytecode noch optimieren. Wir implementieren hier nur Dead Block Elimination. Das bedeutet, dass alle nicht erreichbaren Blöcke aus der Liste der Blöcke gelöscht werden. Eingabe ist Ausgabe des vorherigen Schrittes, Ausgabe ist das gleiche, nur dass die Liste an Blöcken wie eben beschrieben modifiziert wurde.

V. Executing: Im letzten Schritt wird der geparste, gesplittete, verifizierte und optimierte Bytecode nun endlich simuliert.

Schritte I, II und IV sind bereits vollständig implementiert (in den Klassen **PVMParser**, **PVMSplitter** und **PVMOptimizer**). Hier musst du nichts mehr tun. Schritte III und V haben schon Teile der Implementierung vorgegeben, du musst diese allerdings noch geeignet ergänzen (in den Klassen **PVMVerifier** und **PVMExecutable**).

Das Template

Mit dieser Aufgabe erhältst du ein recht großes Code-Template. Lass dich davon nicht abschrecken. Du musst nur in zwei der 16 Dateien etwas ergänzen. Den meisten Code wirst du gar nicht näher ansehen müssen. Im Folgenden wird dir ein kurzer Überblick gegeben, was das Template alles an Code enthält.

Die Klassen der Pipeline

Die fünf Klassen **PVMParser**, **PVMSplitter**, **PVMVerifier**, **PVMOptimizer** und **PVMExecutable** stellen die fünf Schritte der Pipeline dar. Sie haben jeweils eine Hauptmethode, die den Schritt durchführt, sowie Methoden, die auch gleich die nächsten Schritte durchführen.

Die Enums

Die beiden Enums **Type** und **Instruction** listen jeweils alle möglichen Typen von Zellen auf dem Stack und lokalen Variablen respektive alle möglichen Befehle auf.

Die Instruktionen

Die vier Klassen **Instruction**, **IntInstruction** extends **Instruction**, **BlockInstruction** extends **Instruction** und **LabelInstruction** extends **Instruction** bilden eine Hierarchie, die die verschiedenen Instruktionen als Klassen darstellt. **Instructions** speichern dabei nur Art und Zeile der Instruktion, **IntInstructions** sind solche, die noch einen Integer als Parameter nehmen (**LOAD**, **STORE**, **CONST**, **ALLOC**) - diese speichern zusätzlich noch diesen Parameter, **BlockInstructions** sind die beiden Sprung-Befehle - diese speichern noch den Ziel-Block des Sprungs. **LabelInstruction** stellt Labels dar (für Sprung-Befehle), hat aber nur in Pipeline-Schritten I und II Relevanz.

Label

Label stellt Labels dar. Diese Klasse ist für Dich irrelevant.

BasicBlock

Ein **BasicBlock** stellt einen der in Pipeline-Schritt II erzeugten Bytecode-Blöcke dar. Er speichert die in ihm enthaltenen Instruktionen **instructions**, einen Zeiger auf den vorherigen **prev** und nächsten **next** Block (in Standardreihenfolge, also Sprung-Befehle ignorierend). Die **BasicBlocks** bilden also damit eine Doubly Linked List, die dann den gesamten Bytecode repräsentiert.

Zudem speichert jeder Block noch Information über die Blöcke, die ihn über Sprung-Befehle erreichen können (**inbounds**), den Stack von Typen (**initialStack**) und das Array Typen von lokaler Variablen (**initialVariables**), mit denen der Block in Pipeline-Schritt III zum ersten Mal betreten wurde und ein Flag (**reachable**), das speichert, ob der Block überhaupt in irgendeiner Programmausführung erreichbar ist. Diese letzten vier Attribute sind für einzelne Pipeline-Schritte wichtig, müssen Dich aber nicht weiter kümmern.

IO

Das Interface **IO** hat zwei Methoden **read()** und **write()**. Es stellt die Methode dar, mit der die Befehle **READ** und **WRITE** in Pipeline-Schritt V Nutzereingaben anfordern bzw. Ausgaben produzieren. Im Template ist in derselben Datei bereits eine Implementierung von **IO** mitgeliefert (**SystemIO**), die **read()** als Lesen von der Konsole und **write()** als Schreiben auf diese umsetzt und die Du zum Testen verwenden kannst.

PVMError

Dieser Error soll in den unten beschriebenen Situationen geworfen werden.

Main

In dieser Klasse gibt es eine `main()`-Methode, mit der Du deinen Code testen kannst (siehe auch Abschnitt "Test Hilfestellung").

Deine Aufgaben

Klasse `PVMVerifier`

In dieser Klasse sollst Du die `verify()`-Methode an den mit `// TODO` markierten Stellen ergänzen. Diese simuliert den von `entryPoint` aus beginnenden Bytecode nicht mit konkreten Werten der einzelnen Stack-Zellen und Variablen, sondern nur mit deren Typen (`INT`, `BOOL` oder - im Falle von noch nicht initialisierten Variablen - `UNDEFINED`). Wenn dabei mit den Typen oder eventuellen Parametern der Befehle ein Problem auftritt, wird eine Exception geworfen.

Die Methode beginnt beim ersten Block `entryPoint` und simuliert dann solange neue Blöcke, wie diese in das Deque `toVerify` hinzugefügt werden. Dabei kann der gleiche Block mehrfach hinzugefügt und simuliert werden, wenn sich die Typen der lokalen Variablen dabei ändern. Um dies musst Du dir aber keine Gedanken machen. Das ist bereits in der Methode `BasicBlock.incoming()` für Dich implementiert worden. Diese gibt solange `true` zurück, wie der Block, auf dem sie aufgerufen wird, noch nicht fertig bearbeitet ist und erneut in `toVerify` eingefügt werden soll. Um zu wissen, ob ein Block erneut betrachtet werden muss, verwende also diese Methode wie im Template im `case JUMP` im unteren der beiden `switchs` angedeutet.

Für jeden Block, der von `toVerify` gepoppt wird, wird ein `Deque<Type> stack` für den oberen Stack und ein `Type[] variables` für die lokalen Variablen erzeugt. Nun soll jede Instruktion des aktuellen Blocks darauf nacheinander nur mit Typen simuliert werden. Also `LESS` nimmt zweimal `INT` vom Stack und legt `BOOL` darauf usw. Wenn irgendwo auf ein Problem gestoßen wird, also z.B. nicht die korrekten Typen oben auf dem Stack liegen, soll ein `PVMError` geworfen werden. Wenn auf einen an der aktuellen Stelle unzulässigen Befehl gestoßen wird, soll eine `IllegalStateException` geworfen werden. Entscheide selbst, was der Stack bzw. die lokalen Variablen für jeden Befehl für Eigenschaften erfüllen müssen, damit das Programm später unabhängig von den konkreten Eingaben laufen kann. Wirf die entsprechende Exception, wenn dies nicht erfüllt ist und modifiziere die Typen auf dem Stack und die der lokalen Variablen entsprechend, wenn schon.

Am Ende des Blocks muss noch entschieden werden, ob ein weiterer Block/weitere Blöcke in das Deque `toVerify` eingefügt werden müssen. Abhängig vom Befehl kommt entweder nur der nächste Block `block.next` in Frage, oder aber der Block, zu dem ein Jump-Befehl springt. Diese sollen aber - wie bereits erwähnt - nur dann in `toVerify` eingefügt werden, wenn `incoming()` sich auf ihnen zu `true` auswertet.

Auf folgende mögliche Probleme mit dem Bytecode muss geachtet werden und es müssen gegebenenfalls die entsprechenden Exceptions geworfen werden:

- So gut wie jeder Befehl liest den/die obersten Wert(e) vom Stack und benötigt dabei spezielle Typen. Die Typen, die dieser Befehl zum Arbeiten braucht, müssen korrekt sein.
- Manche Befehle nehmen zusätzlich noch Parameter. Diese dürfen nicht out of bounds sein. (Bezieht sich *NICHT* auf die Parameter der JUMP-Befehle!)
- Manche Befehle dürfen nur in bestimmten Positionen in einem Block vorkommen (oder dürfen überhaupt nicht auftauchen).
- Wenn man am Ende eines Blocks nicht wegspringt oder anhält, wird in den nächsten Block gegangen. Dieser muss dann natürlich existieren.

Hinweise:

- Für diese Aufgabe mag [diese Spezifikation](#) unseres Bytecode-Dialektes hilfreich sein.
- Sieh Dir die Hilfsmethoden `PVMVerifier.checkSize()`, `PVMVerifier.tryPop()` und `PVMVerifier.tryPeek()` an. Sie könnten hilfreich sein.

Klasse `PVMExecutable`

In dieser Klasse sollst Du die `run()`-Methode an den mit `// TODO` markierten Stellen ergänzen. Ziel ist es, den durch das Attribut `entryPoint` beschriebenen Bytecode Block für Block (eingestiegen wird logischerweise beim Block `entryPoint`, welcher Block nach diesem kommen soll, musst du dir dann selbst überlegen) zu simulieren. Vorgegeben ist der Stack, aufgeteilt in die lokalen Variablen `variables` und den oberen, restlichen Teil des Stacks `stack`. Der `stackPtr` zeigt auf die aktuell oberste verwendete Stelle im oberen Stack. Du musst nun für jede mögliche der 27 erlaubten Instruktionen in Abhängigkeit vom aktuellen Stack, den lokalen Variablen, dem Parameter des Befehls, falls vorhanden und einer eventuellen Nutzereingabe den Stack, die lokalen Variablen und den aktuellen Block jeweils passend modifizieren (natürlich i.d.R. nur eines/einige davon) und eventuell eine Ausgabe produzieren. Für Ein- und Ausgaben (`READ`, `WRITE`) verwende die Befehle `io.read()` und `io.write()` des übergebenen `IO`-Objektes. Eine Implementierung, die von der Konsole einliest und auf dieser ausgibt, ist Dir mitgeliefert worden, sodass du deinen Code leicht testen kannst.

Hinweis: Du darfst hier davon ausgehen, dass die Code-Blöcke alle korrekt sind, also dass stets zum Anfang eines Code-Blocks hin und stets nur vom Ende eines Code-Blocks aus gesprungen wird, da dies ja bereits in Schritt III verifiziert wurde.

Test Hilfestellung

- Es ist eine Main Klasse enthalten. Im `minijvm` Ordner abgelegte Dateien können darin wie im Beispiel gezeigt genutzt werden.
- Es ist eine Beispiel Test Klasse unter `test` enthalten. Darin können Dateien sowohl aus `minijvm` als auch aus `test` benutzt werden. Bei Namenskonflikten hat `test` Priorität.
- Alle in `minijvm` enthaltenen Dateien werden von diesem Test verifiziert. Damit kann man überprüfen, ob der Bytecode korrekt ist.

minijvm verify test

Dieser Test schlägt immer fehl, das muss so. Artemis kann nur bei fehlschlagenden Tests weitere Informationen anzeigen.

-  Dateien in minijvm 0 of 1 tests passing

FAQ


- **Frage:** Der Stack in der `PVMExecutable.run()`-Methode ist ein `int`-Array. Es müssen darauf aber teils auch `booleans` gespeichert werden. Wie stelle ich diese dar?

Das darfst und musst du dir selbst überlegen.

[Musterlösung](#)

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. 

[About](#)

[Request change](#) [Release notes](#) [Privacy Statement](#) [Imprint](#)