

THE UNIVERSITY OF MELBOURNE
SWEN90006: SOFTWARE TESTING AND RELIABILITY

Assignment 1

SECOND SEMESTER, 2018

DUE DATE: 18:00PM, SUNDAY, 27 AUGUST, 2017

Introduction

The first assignment deals with input partitioning, boundary-value analysis, and control-flow testing. You are given a specification and a program that implements that specification. The aim of this assignment is to test the program using the different techniques, and to analyse the difference between them.

You are expected to derive and compare test cases, but you are not expected to debug the program.

The assignment is part laboratory exercise because you are expected to write a JUnit driver program to run your test cases automatically. Some exploration may be needed here. The assignment is also part analysis exercise as you are expected to apply the testing techniques to derive your test cases, and to compare them. Finally, the assignment is also part competition, as your solutions to various tasks will be evaluated against all other submissions to measure its effectiveness and completeness.

The assignment is worth 20% of your final mark.

Specification: The Java Machine

Java code is compiled by `javac` into *bytecode*. The bytecode is then executed by a *virtual machine* called the Java Virtual Machine (JVM). The JVM¹ provides many of Java's safety guarantees.

The system-under-test is a Java program that reads an input file that contains a program written in a simple assembly language and executes the given assembly language program. At the end of executing the program successfully, it prints the program's final result.

The Assembly Language

The VM accepts input programs from text files (assuming UTF-8 encoding²). Programs written in the assembly language of the VM are a series of lines where each line contains a single instruction. The assembly language is case-insensitive. “;” begins a single line comment. Blank lines and those on which only comments appear are interpreted as instructions that do nothing (“*no-ops*”).

¹Assisted by Java's type checker

²It might throw a `java.nio.charset.MalformedInputException` if otherwise.

```

1      ;; factorial program, to calculate N!
2
3      ;; global constants:
4      ;;   R3 holds 'N', the thing we are computing factorial of
5      ;;   R2 holds 1, the increment value used below
6      MOV R3 12                ; N = 12
7      MOV R2 1                ;
8
9      ;; local variables
10     ;;   R1 holds 'i', which is a counter from 0 .. N
11     ;;   R0 holds 'n', which is always equal to i!
12     MOV R1 0                ; i = 0;
13     MOV R0 1                ; n = 1;
14
15     ;; program body
16     ;; loop invariant (see SWEN90010 next semester): n = i!
17     SUB R4 R3 R1            ; while(i != N)
18     JZ  R4 4                ; {
19     ADD R1 R1 R2            ;   i = i + 1;
20     MUL R0 R0 R1            ;   n = n * i;
21     JMP -4                 ; }
22     RET R0                 ; return n;

```

Figure 1: An example assembly program for computing factorials.

The assembly language supports 10 instructions, each of which is a member of the following grammar, written here in *Backus-Naur Form* (BNF) style notation.³

```

<INSTRUCTION> ::=
    "add" <REGISTER> <REGISTER> <REGISTER>
    | "sub" <REGISTER> <REGISTER> <REGISTER>
    | "mul" <REGISTER> <REGISTER> <REGISTER>
    | "div" <REGISTER> <REGISTER> <REGISTER>
    | "ret" <REGISTER>
    | "mov" <REGISTER> <VALUE>
    | "ldr" <REGISTER> <REGISTER> <VALUE>
    | "str" <REGISTER> <VALUE> <REGISTER>
    | "jmp" <VALUE>
    | "jz" <REGISTER> <VALUE>

```

Here “<REGISTER>” denotes a register name and “<VALUE>” a decimal integer value (e.g. 0, 325, -1027 etc.).

The assembly language includes instructions like “ldr” and “str” for reading and writing to memory. The virtual machine has a memory of size 65536 words, where each word is a (32-bit, signed) Java `int`.

The virtual machine has a special piece of state called the *program counter* (pc) which tracks which instruction (i.e. which line of the assembly program) it is currently executing. Certain instructions like “jmp” and “jz” can be used to jump to different parts in the program, and so

³https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form

explicitly modify the pc. Otherwise, each instruction executes in order (i.e. after executing any non-“jmp” and non-“jz” instruction, the pc is always incremented by 1, so that the instruction on the next line is executed).

If Ra , Rb and Rc are register names and val is an integer value, then the behaviour of each instruction is as follows:

Instruction	Description
ADD $Ra\ Rb\ Rc$	Adds the values in registers Rb and Rc and puts the result into Ra . We abbreviate this behaviour using the notation: $Ra = Rb + Rc$
SUB $Ra\ Rb\ Rc$	$Ra = Rb - Rc$
MUL $Ra\ Rb\ Rc$	$Ra = Rb * Rc$
DIV $Ra\ Rb\ Rc$	$Ra = Rb / Rc$
RET Ra	Causes the program to finish, returning whatever value is in Ra .
MOV $Ra\ val$	Puts the value val into register Ra .
LDR $Ra\ Rb\ val$	If Rb holds some value b , then first compute $b + v$. Then read from memory the value at address $b + v$ and put this into Ra .
STR $Rb\ val\ Ra$	If Rb holds some value b , then first compute $b + v$. Then store the value in register Ra into memory at the address $b + v$.
JMP val	Alter the pc by adding val to it. So “JMP 0” is an infinite loop, and “JMP 1” does nothing (i.e. is a no-op).
JZ $Ra\ val$	If register Ra holds the value zero, alter the pc by adding val to it. (Otherwise, increment the pc by 1.)

The LDR and STR instructions are used to read from and write to memory respectively. The example program `array.s` in the `examples/` directory shows these instructions in action. It creates an array with the integers 0 ... 9 and then iterates over the array, reading each of its elements and summing them together to produce the final result 45.

Executing a jump instruction (i.e. “jmp” or “jz”) that causes the pc to become negative or to become greater than $N - 1$, where N is the number of lines (i.e. instructions) in the input program, causes the program to finish early without returning a value.

The assembly language has 32 registers, named R0, R1, R2, ..., R31. Each register holds an (32-bit, signed) Java `int` value.

An assembly program is valid when all of the following conditions hold:

- each instruction adheres to the grammar above (i.e. is syntactically correct);
- each register mentioned in the program is valid (i.e. is one of R0 to R31);
- each value val mentioned in each instruction is no smaller than -65535 and no larger than 65535; and
- The last instruction executed by the program is always a RET instruction, to ensure that it returns a value.

Exceptions If given a program that does not satisfy these conditions, the VM must throw either the `InvalidInstructionException` (if one of the first three conditions does not hold), or the `NoReturnValueException` (if the final condition does not hold).

If given a valid program (i.e. one that does satisfy all of these conditions), a bug-free VM should execute successfully producing a final return value for the program or looping forever (if the assembly program contains an infinite loop).

Inputs The function has just one input: a list of strings, each containing a command.

Requirements The function has the following requirements:

1. The machine must execute the input instructions in order from left to right, unless encountering a JMP instruction.
2. The table above specifies the behaviour of each of the instructions supported by the machine. Each instruction must implement the behaviour outlined in this table.
3. If an error occurs when executing an instruction in the list, execution must be halted at that point and an exception must be raised. These exceptions are outlined in the table above.
4. The *result* of program is the first “ret” instruction executed.
5. All bytes other than those listed in the table above are unknown instructions and attempting to execute one of them must throw the exception `InvalidInstructionException`.

Assumptions The function makes the following assumptions about the input:

1. Programs adhere to the grammar – that is, it can be assumed that there are no syntax errors.

NOTE: this does not mean that each program is *valid*; just that each program is syntactically correct. Other exceptions can be thrown.

2. The number of instructions is assumed to be greater than 0.

The Program

To obtain the source for the program, fork the repository at:

<https://gitlab.eng.unimelb.edu.au/tmiller/SWEN90006-A1-2018.git>

See the handout on the LMS for instructions on creating forks, pulling & pushing changes, and committing changes.

NOTE: If you find any faults in the implementation, please let us know via the discussion board. We will correct the fault and ask that everyone pull changes. There are not intended to be faults in the implementation, but software engineering is hard!

Building and Running the Program

The VM code lives in the `src/swen90006/machine` directory. The top-level file is `SimpleDriver.java`, and so it can be built by running “`javac *.java`” in the `machine/` directory.

It has been successfully built and tested on Java 1.8 but should also work with some earlier versions of Java.

The file `build.xml` contains an Ant build script, should you choose to use it.

There are two JUnit test scripts in `test/swen90006/machine`. You will need to modify each of these to complete the tasks below. You can run these by compiling and running as a class, but you will need to include the library files in the `lib/` directory. You can also run the JUnit test scripts running `ant partitioning` and `ant boundary` respectively.

Tasks and Questions

Task 1

Using the specification, apply equivalence partitioning to derive equivalence classes for the input domain of the `execute` method.

Document your equivalence partitioning process using a test template tree. You will be marked *only* on your test template tree, so ensure that it is clear and concise. You can omit some nodes to improve readability, provided that it is clear what you intend. For example, in tutorial 2, if I wanted to test all 12 months, I would create nodes for JAN and DEC, and then use “...” in between them to represent the other months.

Do your set of equivalence classes cover the input space? Justify your claim.

Task 2

Select test cases associated with your equivalence classes, and implement these in the JUnit test driver named `test/swen90006/machine/PartitioningTests.java`. For each test, clearly identify from which equivalence class it has been selected.

Task 3

Conduct a boundary-value analysis for your equivalence classes. Show your working for this. Select test cases associated with your boundary values.

Task 4

Implement your boundary-value tests in the JUnit test driver called `test/swen90006/machine/BoundaryTests.java`.

You may choose to implement multiple tests (for the same equivalence class) in the same JUnit test. For each test, clearly identify the boundary (i.e. between which equivalence classes) are being tested.

Task 5

Calculate the coverage score of your two test suites (equivalence partitioning and boundary-value analysis) using *multiple-condition coverage* on *only* the `execute` method. That is, there is no

requirement to measure coverage of any other method, even if called by the `execute` methods. Show your working for this coverage calculation.

NOTE: You do NOT need to draw the control-flow graph for your solution.

Task 6

Derive five *non-equivalent* mutants for the machine class using the mutation operators in the notes, and that you believe will be difficult to find using testing. Insert each of these mutants into the files `mutants/mutant-1/swen90006/machine/`, `mutants/mutant-2/swen90006/machine/`, etc.

It is important that these mutants are both non-equivalent AND that each mutant is killed by at least one test in your JUnit script to demonstrate that they are non-equivalent.

Importantly, do not change anything else about the mutant files except for inserting the mutant.

Question 7

Compare the two sets of test cases (equivalence partitioning and boundary-value analysis) and their results. Which method did you find was more effective and why? You should consider the coverage of the valid input/output domain, the coverage achieved, and the mutants it kills. Limit your comparison to half a page. If your comparison is over half a page, you will be marked only on the first half page.

Marking criteria

As part of our marking, we will run your JUnit scripts on everyone else's mutants. You will receive marks for killing other mutants as well as for deriving mutants that are hard to kill. This will contribute 5 marks to the total.

Criterion	Description	Marks
Equivalence partitioning	Clear evidence that partitioning the input space to find equivalence classes has been done systematically and correctly. Resulting equivalence classes are disjoint and cover the appropriate input space	5
Boundary-value analysis	Clear evidence that boundary-value analysis has been applied systematically and correctly, and all boundaries, including on/off points, have been identified	5
Control-flow analysis	Clear evidence that measurement of the control-flow criterion has been done systematically and correctly	2
	There is a clear and succinct justification/documentation of which test covers each objective	2
Discussion	Clear demonstration of understanding of the topics used in the assignment, presented in a logical manner	1
JUnit tests	JUnit scripts implement the equivalence partitioning and boundary-value tests, and find many mutants	2.5
Mutants	Selected mutants are valid mutants and are difficult to find using tests	2.5
Total		20

For the JUnit tests, the score for these will be calculated using the following formula:

$$junit_score = \frac{\frac{k}{T}}{\ln(N) + 10}$$

in which N is the number of tests in your test suite, k is the number of mutants that your test suite kills, and T is the maximum number of mutants killed by any other JUnit test suite⁴. The entire pool of mutants are the mutants from all other submissions. Therefore, your score is the mutant score, divided by $\ln(N) + 10$, which incentivises smaller test suites⁵.

For the mutants, the score is:

$$mutant_score = \frac{\sum_i^M \sum_j^N a_{i,j}}{T}$$

in which M is the total number of your mutants, N is the total number of other people's test suites, $a_{i,j} = 1$ if mutant i is still alive after executing test suite j , and $T \leq M + N$ is the highest number of mutants still alive by any student in the class. This is then scaled to be out of 2.5. Therefore, your score is the inverse of the mutant score of all other students' test suites on your mutants, which incentivises you to submit hard-to-find mutants, while T normalises the score to ensure that everyone is rewarded for good mutants.

⁴This ensures that equivalent mutants are not counted.

⁵This incentive is to resist the urge to submit a test suite of thousands of tests with the hope of increasing the score.

Submission

For the JUnit test scripts, we will clone everyone's Gitlab repository at the due time. We will mark the latest version on the *master* branch of the repository. To have any late submissions marked, please email Tim to let him know to re-clone your repository.

Some important instructions:

1. Do NOT change the package names in any of the files.
2. Do NOT change the directory structure.
3. In short: you should be able to complete the assignment without add any new source files.

JUnit scripts will be batch run automatically, so any script that does not follow the instructions will not run.

The remainder of the assignment (test template tree, boundary-value analysis working, coverage, and discussion) submit a PDF file using the Turnitin links on the subject LMS. Go to the SWEN90006 LMS page, select *Assessment* from the subject menu.

Tips

Some tips to managing the assignment, in particular, the equivalence partitioning:

1. Ensure that you understand the notes *before* diving into the assignment. Trying to learn equivalence partitioning or boundary-value analysis on a system this size is difficult.
2. Keep it simple: don't focus on what you think we want to see — focus on looking for good tests and then documenting them.
3. Focus on the requirements: as with any testing effort, focus your testing on the requirements, NOT on demonstrating the theory from the notes. If you are worrying about whether to apply a particular part of the theory, look at whether it tests one of the requirements. If not, it is probably not useful.

Academic Misconduct

The University misconduct policy⁶ applies. Students are encouraged to discuss the assignment topic, but all submitted work must represent the individual's understanding of the topic.

The subject staff take academic misconduct very seriously. In this subject in the past, we have successfully prosecuted several students that have breached the university policy. Often this results in receiving 0 marks for the assessment, and in some cases, has resulted in failure of the subject.

⁶See <https://academichonesty.unimelb.edu.au/>