TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

Student ID: .....

EXAMINATIONS — 2016

TRIMESTER 2

SWEN224

Software Correctness

Time Allowed: ONE HOUR

Instructions: Answer all questions
All questions are of equal value

Answer all questions in the boxes provided.
Every box requires an answer.
If additional space is required you may use a separate answer booklet.

No calculators permitted.
Non-electronic Foreign language to English dictionaries are allowed.

Question	Topic	Marks
1.	Static Analysis	20
2.	Specification	20
3.	Loop Invariants	20
	Total	60

Question 1. Static Analysis [20 marks]

(a) [3 marks] Static analyses operate at *compile time*. Briefly, discuss the disadvantages of this, compared with techniques which operate at *run time*.

Operating at compile time means that all possible inputs are considered when looking for specific problems in the code, and we can make strong guarantees that certain problems do not occur. In contrast, operating at runtime means that only a small number of inputs can be considered, and we cannot be sure problems won't arise with different inputs.

However, operating at compile time means static analysis is necessarily conservative and may report "false-positives". In contrast, runtime checking always reports true errors.

(b) This question is concerned with *definite assignment* in Java.

(i) [4 marks] Briefly, explain what definite assignment means in Java. You may illustrate your answer with examples as necessary.

Definite assignment is the process used in Java for checking that a variable is guaranteed to be assigned a value before it is ever used. The process is approximate and operates over the control-flow graph of a method. The following illustrates a program that fails definite assignment:

```
1       public int f(int x) {
2           int r;
3           //
4           if(x >= 0) {
5               r = x;
6           }
7           //
8           return r;
9       }
```

This program fails because the variable `r` is not definitely assigned when the **`return`** statement is reached.

Consider the following method in Java:

```
1       int max(int[] xs) {
2           if (xs.length == 0) {
3               throw new IllegalArgumentException("invalid array");
4           } else {
5               int r;
6               for (int i = 0; i != xs.length; ++i) {
7                   if (i == 0 || xs[i] > r) {
8                       r = xs[i];
9                   }
10              }
11              return r;
12          }
13      }
```

(ii) [2 marks] Briefly, explain why the above method fails definite assignment.

The above method fails definite assignment because Java cannot be sure that the variable `r` is assigned a value before it is used. For example, Java will consider the loop may not execute any iterations so that **`return`** `r` is executed immediately after **`int`** `r`.

(iii) [5 marks] Like most static analyses, definite assignment is said to be *conservative*. Explain what this means using the above `max(int[])` method to illustrate.

In the example above, Java cannot tell that variable `r` is always assigned before being used. However, we as programmers can (by careful reasoning) see that it always is. Thus, in this case, Java is reporting an error for a program that is actually correct. This is behaviour is said to be *conservative*, as Java may report errors which are *false positives* (i.e. not real). However, conservatism also implies that it should not miss any errors which actually exist (i.e. no *false negatives*).

(c) [6 marks] For each *parameter*, *return* and *field* in the following program, insert `@NonNull` or `@Nullable` annotations (where appropriate) by writing in the box.

```
1  public class Tree {
2      private @NonNull Object data; // Every tree node has a data item
3
4      private @Nullable Tree left;    // Some trees have a left child
5
6      private @Nullable Tree right;   // Some trees have a right child
7
8      public Tree(@NonNull Object data) {
9          if (data == null) {
10             throw new IllegalArgumentException("Data cannot be null");
11         }
12         this.data = data;
13
14         this.left = null;
15
16         this.right = null;
17     }
18
19     public @NonNull Object getData() {
20         return data;
21     }
22
23     public @Nullable Tree getLeftChild() {
24         return left;
25     }
26
27     public @Nullable Tree getRightChild() {
28         return right;
29     }
30
31     public void appendLeft(@NonNull Object item) {
32         if (left == null) {
33             left = new Tree(item);
34         } else {
35             left.appendLeft(item);
36         }
37     }
38
39     public void appendRight(@NonNull Object item) {
40         if (right == null) {
41             right = new Tree(item);
42         } else {
43             left.appendRight(item);
44         }
45     }
46 }
```

Student ID: .....

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Question 2. Specification [20 marks]

(a) For each of the following, provide one set of parameter values which meet the precondition and one set which does not.

(i) [2 marks]

```
function decrement(int value) -> (int r)
requires value > 0:
    //...
```

Meets precondition: `value=1`

Does not meet precondition: `value=0`

(ii) [2 marks]

```
function copy(int[] xs, int start, int[] ys) -> (int[] r)
requires (start+|xs|) < |ys| && start >= 0:
    //...
```

Meets precondition: `xs=[0],start=0,ys=[1,2]`

Does not meet precondition: `xs=[0],start=0,ys=[1]`

(iii) [2 marks]

```
function findNegative(int[] xs) -> (int r)
requires some { i in 0..|xs| | xs[i] < 0 }:
    //...
```

Meets precondition: `xs=[0,1,0,-1]`

Does not meet precondition: `xs=[1,2,3]`

(iv) [2 marks]

```
function bitwiseAnd(int[] lhs, int[] rhs) -> (int[] r)
requires all { i in 0..|lhs| | 0 <= lhs[i] && lhs[i] <= 255 }
requires all { i in 0..|rhs| | 0 <= rhs[i] && rhs[i] <= 255 }:
    //...
```

Meets precondition: `lhs=[76,111,118,101],rhs=[87,104,105,108,101,121]`

Does not meet precondition: `lhs=[-2,-1,0,1,2],rhs=[32]`

(b) Consider the following implementation for the function `find()`:

```
1 function find(int n, int[] items, int i) -> (int r):
2     if i == |items|:
3         return i
4     else if items[i] == n:
5         return i
6     else:
7         return find(n,items,i+1)
```

(i) [4 marks] Briefly, describe in your own words what function `find()` does.

`find()` searches through the `items` array starting from `i` looking for the first occurrence of `n`. If one is found, its index is returned; otherwise, the length of `items` is returned.

(ii) [8 marks] Provide an appropriate specification for function `find()`.

```
requires i >= 0 && i <= |items|
ensures r >= i && r <= |items|
ensures r != |items| ==> items[r] == n
ensures all { k in i..r | items[k] != n }
```

Question 3. Loop Invariants [20 marks]

(a) [6 marks] Briefly, discuss what a *loop invariant* is and explain the *three rules* of loop invariants.

A loop invariant is a property maintained throughout the execution of a loop. Loop invariants can be helpful for understanding loops and reasoning about their behaviour. Loop invariants are not part of a function's specification, but are needed by the verified to ensure a function meets its specification.

The three rules of loop invariants are:

1. **Loop invariants must hold on entry**. That is, they must hold before the first iteration of the loop begins.
2. **Loop invariants must be restored after each iteration**. That is, they must hold at the end of each iteration of the loop (assuming they held at the start of each iteration).
3. **Loop invariants must hold on exit**. This follows from rule 2 but, in addition, when the loop condition is false, the loop invariant must still hold.

Consider the following implementation for the function `count()`:

```
1 function count(int from, int to) -> (int r)
2 requires from < to:
3     //
4     int i = from
5     //
6     while i < to where i >= 0:
7         i = i + 1
8     //
9     return i
```

(b) [3 marks] Briefly, discuss whether or not the given loop invariant is correct.

The loop invariant is incorrect as it does not hold on entry (rule 1 above). This is because no restriction is placed on the possible values of variable `from`. Thus, `from` could be negative and, hence, `i` could be negative at the start of the loop.

Consider the following implementation for the function `rotate()`:

```
1  function rotate(int[] xs) -> (int[] ys)
2  requires |xs| > 0:
3  ensures |xs| == |ys|
4  ensures all { k in 1 .. |xs| | xs[k] == ys[k-1] }
5  ensures ys[|xs|-1] == xs[0]:
6      //
7      int first = xs[0]
8      int[] ys = xs
9      int i = 1
10     //
11     while i < |xs|:
12         ys[i-1] = xs[i]
13         i = i + 1
14     //
15     ys[|xs|-1] = first
16     //
17     return ys
```

(i) [5 marks] Briefly, describe in your own words what function `rotate()` does.

The `rotate()` function moves every element in the `xs` array one position to the "left" (i.e. to the index below). The element at the start of the array is "wrapped around" to become the last element.

(ii) [6 marks] Provide an appropriate *loop invariant* for function `rotate()`.

```
where i > 0 && |xs| == |ys|
where all { k in 1 .. i | xs[k] == ys[k-1] }
```

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Student ID: .....

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

****