Assignment 1

Full name: Yichu Li
Student number: 20817846
Email: y2792li@uwaterloo.ca

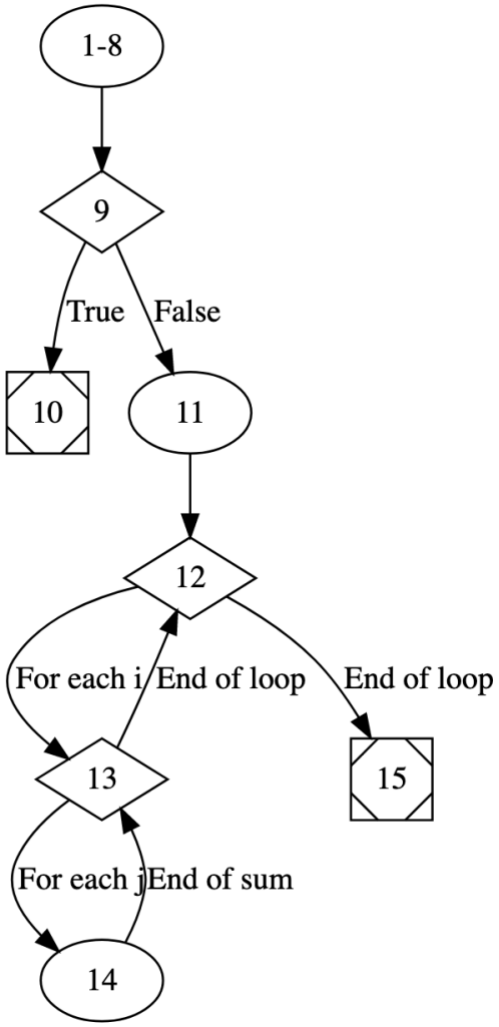
Question 1
(a)
Test case: a = None, b = [1,2]
A null value in a can result in an IndexError when it is trying to find the length of a "None" object in line 7, before the code reaches the fault in line 8. Therefore, there is no execution of the fault.

(b)
Test case: a = [[1], [2]], b = [[3]]
The value of p1 and q are the same value of 1. This does not cause an error from the test case. However, fault will occur since line 8 will be executed.

(c)
Test case: a = [[1, 2, 3], [4, 5, 6]], b = [[1], [2], [3], [4]]
The test case will result in an error since the value of p1 and q should be 4 and 1, respectively. However, the value of p1 and q is 1 and 4, which lead to an error. It will not result in a fault since it will raise ValueError because of incompatible dimensions in line 9. The actual output is as expected.

(d)
First error state:
a: [[5, 7], [8, 21]]
b: [[8], [4]]
n: 2
p: 2
q: 2
p1: 1
c: undefined
i: undefined
j: undefined
k: undefined
PC: after q, p1 = len(b), len(b[0])

(e)



Question 2

(a)

```
class Ast(object):
    """Base class of AST hierachy"""
    pass

class Stmt(Ast):
    """A single statement"""
    pass

class AsgnStmt(Stmt):
    """An assignment statement"""
    def __init__(self, lhs, rhs):
        self.lhs = lhs
        self.rhs = rhs

class IfStmt(Stmt):
    """If-then-else statement"""
    def __init__(self, cond, then_stmt, else_stmt=None):
        self.cond = cond
```

```
        self.then_stmt = then_stmt
        self.else_stmt = else_stmt

class RepeatUntilStmt(Stmt):
    """Repeat-until statement"""
    def __init__(self, repeat_stmt, cond):
        self.repeat_stmt = repeat_stmt
        self.cond = cond
```

(b)
Formalization:
There are two rules for this.
Rule 1: When Repeat-Until b evaluates to be true:

$$\frac{< S, q >\Downarrow q' \quad < b, q' >\Downarrow \textbf{\textit{True}}}{< \textbf{\textit{repeat}}\ S\ \textbf{\textit{until}}\ b, q >\Downarrow q'}$$

Rule 2: When Repeat-Until b evaluates to be false:

$$\frac{< S, q >\Downarrow q' \quad < b, q' >\Downarrow \textbf{\textit{False}} \quad < \textbf{\textit{repeat}}\ S\ \textbf{\textit{until}}\ b,\ q' >\Downarrow q''}{< \textbf{\textit{repeat}}\ S\ \textbf{\textit{until}}\ b, q >\Downarrow q''}$$

(c)
Here is the derivation tree that I constructed using the rules:

$$\dfrac{< x := 2, [] >\Downarrow [x := 2] \quad \dfrac{< x := x - 1, [x := 2] >\Downarrow [x := 1] \quad < x \leq 0, [x := 1] >\Downarrow \textbf{\textit{False}} \quad \dfrac{< x := x - 1, [x := 1] >\Downarrow [x := 0] \quad < x \leq 0, [x := 0] >\Downarrow \textbf{\textit{True}}}{< \textbf{\textit{repeat}}\ x := x - 1\ \textbf{\textit{until}}\ x \leq 0, [x = 1] >\Downarrow [x := 0]}}{< \textbf{\textit{repeat}}\ x := x - 1\ \textbf{\textit{until}}\ x \leq 0, [x := 2] >\Downarrow [x := 0]}}{< x := 2; \textbf{\textit{repeat}}\ x := x - 1\ \textbf{\textit{until}}\ x \leq 0, [] >\Downarrow [x := 0]}$$

(d)
We can have two rules for "$S\ ;\ \textbf{\textit{if}}\ b\ \textbf{\textit{then skip else}}\ (\textbf{\textit{repeat}}\ S\ \textbf{\textit{until}}\ b)$":
Rule 1: When If-Skip-else b evaluates to be true:

$$\frac{< S, q >\Downarrow q' \quad < b, q' >\Downarrow \textbf{\textit{True}}}{< S; \textbf{\textit{if}}\ b\ \textbf{\textit{then skip else}}\ (\textbf{\textit{repeat}}\ S\ \textbf{\textit{until}}\ b), q >\Downarrow q'}$$

Rule 2: When If-Skip-else b evaluates to be false:

$$\frac{< S, q >\Downarrow q' \quad < b, q' >\Downarrow \textbf{\textit{False}} \quad < \textbf{\textit{repeat}}\ S\ \textbf{\textit{until}}\ b, q' >\Downarrow q''}{< S; \textbf{\textit{if}}\ b\ \textbf{\textit{then skip else}}\ (\textbf{\textit{repeat}}\ S\ \textbf{\textit{until}}\ b), q >\Downarrow q''}$$

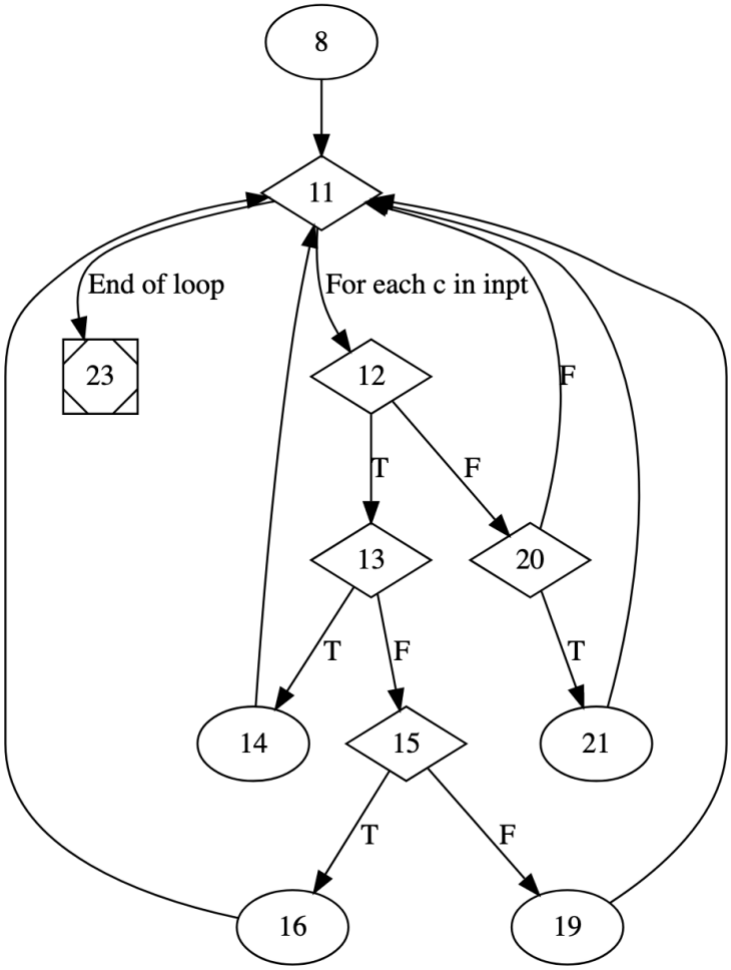
According to (b), these two rules are the same as Repeat-Until when b evaluates to be true and false, respectively. When b evaluates to true, both constructs terminate and result in the same final state q'; when b evaluates to false, both constructs continue and lead to the same final state q''.
Therefore, the statement

$$\textbf{\textit{repeat}}\ S\ \textbf{\textit{until}}\ b$$

is semantically equivalent to

$$S\ ;\ \textbf{\textit{if}}\ b\ \textbf{\textit{then skip else}}\ (\textbf{\textit{repeat}}\ S\ \textbf{\textit{until}}\ b)$$

Question 3
(a)



(b)

According to the written CFG, we can list the sets of TRs.

For node coverage (NC), TR contains each reachable node in the CFG from (a).

TR_{NC} ={8,11,12,13,14,15,16,19,20,21,23}

For edge coverage (EC), TR contains each reachable path of length up to 1, inclusive, in the CFG.

TR_{EC} = {[8,11], [11,12], [11,23], [12,13], [12,20], [13,14], [13,15], [14,11], [15,16], [15,19], [16,11], [19,11], [20,21], [20,11], [21,11]}

Infeasible TR for edge coverage (EC): {[20,11]}. One path from the TR is infeasible as there is no possible scenario to go through the path [20,11]. Because the state can only become either 0 or 1 and the "elif" in line 20 cannot be false.

For edge-pair coverage (EPC), TR contains each reachable path of length up to 2, inclusive, in the CFG from (a).

TR_{EPC} = {[8,11,12], [8,11,23], [11,12,13], [11,12,20], [12,13,14], [12,13,15], [12,20,11], [12,20,21], [13,14,11], [13,15,16], [13,15,19], [14,11,23], [14,11,12], [15,16,11], [15,19,11], [16,11,12], [16,11,23], [19,11,12], [19,11,23], [20,11,12], [20,11,23], [20,21,11], [21, 11, 12], [21,11,23]}

Infeasible TR for edge-pair coverage (EPC): {[12,20,11], [20,11,12], [20,11,23]}. The reason is the same as the edge coverage. The path [20,11] is impossible to go through. Because the variable

state can only become either 0 or 1 and the “elif” in line 20 cannot be false. Therefore, all the edge-pair that contains path [20,11] are infeasible.

(c)
Node Coverage but not Edge Coverage is infeasible. Because covering all nodes would naturally lead to covering all edges as well, making it infeasible to have full node coverage without full edge coverage. In the code, each decision point leads directly to a unique node, and we must traverse the edge to that node to cover it. Thus, covering all nodes will naturally cover all edges in this case.

Question 4
(a)

File	Coverage
ast.py	100%
int.py	91%
parser.py	87%

ast.py is fully covered.

int.py:75
Reason: the assert False will not be execute as node.op can only be "<=", “<”, “=”, “>=” and “>” in the visit_RelExp function.
int.py:179-184
Reason: because we did not execute the int.py as a main program, as the _parse_args() is executed in the main program.
int.py: 188-193, 196-197:
Reason: because we didn’t execute the int.py as a main program.

parser.py: 29, 40, 41
Reason: because we did not invoke the WhileLangBuffer class.
parser.py: 481, 482
Reason: because we did not use to parse a newline.
parser.py: 487-586
Reason: because we did not invoke the WhileLangSemantics class.
parser.py: 604-609
Reason: because we didn’t execute the parser.py as a main program.

(b)
ast.py is fully covered.

int.py: 72
Reason: line 72 will always be true because node.op can only be "<=", “<”, “=”, “>=” and “>” in the visit_RelExp function.
int.py: 90
Reason: line 90 will always be true as it has listed all possible value for node.op.
int.py: 111

Reason: line 111 will always be true as it has listed all possible value for node.op.
int.py: 196
Reason: because we didn’t execute the int.py as a main program.

parser.py: 603
Reason: because we didn’t execute the parser.py as a main program.

Conclude

The interpreter is likely to be functionally correct for the set of inputs and conditions that have been tested. This is a significant achievement and boosts confidence in the software.
With high code coverage, we can minimize the risk of undetected bugs in the parts of the code that have been tested. Apart from that, to achieve maintainability, a complete, comprehensive test suite makes it easier to refactor and extend the code in the future.