Moti Begna

CSCI 5106

**Homework 3 – Rework(1, 4, 5, 6)**

Problem 1:

a) To prove the correctness of the program, the ideology of invariants/Hoare’s Triples can be utilized to determine if certain assertions hold at any point of the program at runtime. Essentially, if a given precondition is true before the execution of line, and that line executes, then the post-condition must also be true as well.

b) Let the precondition for this program be {M(x) is a new value}, let the program line(s) be:

write(M(x));

while M(x) – M(x+1) >= 0 and M(x+1) – M(x) >=0

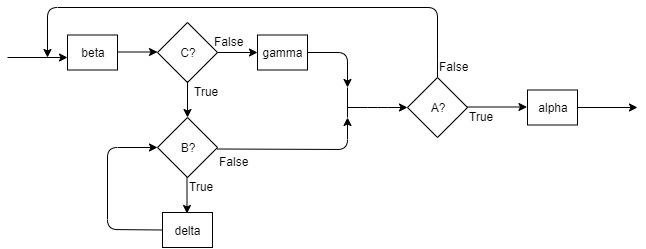
x = x + 1;

Let the post condition be that {M(x) is not a duplicate of the previous value}. Here, we can see that if the precondition holds, then the program lines (which are essentially the RAM program) would ensure that the postcondition is also true. Thus, given a sequence of values, the program would correctly write out the sequence with adjacent duplicates removed.

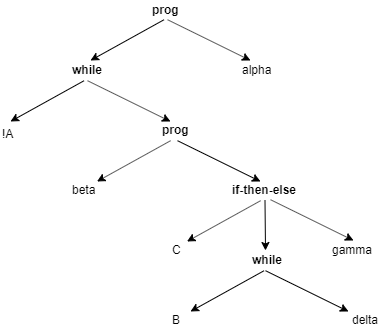
c) The reason why the first program on page 6 of the text is more difficult in explaining its correctness has to do with how it checks whether or not adjacent values are equal. While the program on page 68 simply checks if the next value is equal to the first value in the “run”, the former program relies on subtracting the values from each other—thus, if they are equal the result would always be 0. This adds a level of abstraction that is not immediately comprehendible at first.

Problem 2:

a)

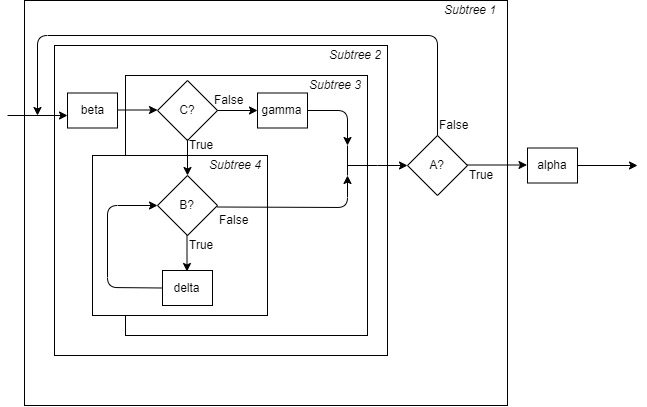
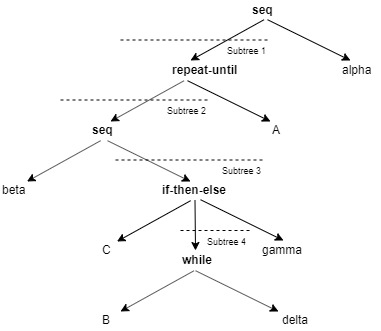


Flow Diagram



Abstract Syntax Tree

b)



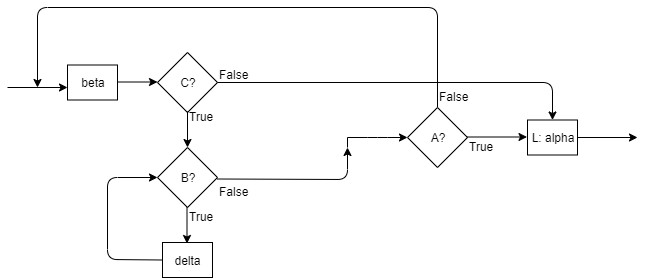
Subtree 1 : 1 edge in, 1 edge out

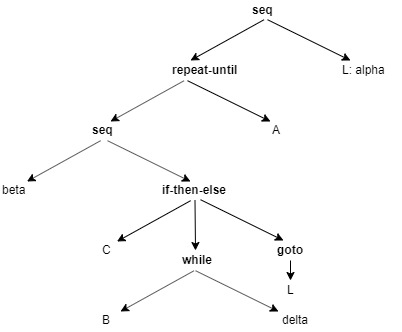
Subtree 2: 1 edge in, 1 edge out

Subtree 3: 1 edge in, 1 edge out

Subtree 4: 1 edge in, 1 edge out

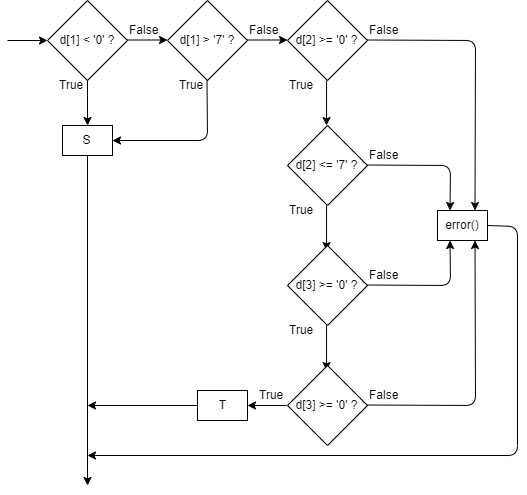
c)





d) The subparts between the AST and the flow diagram would remain similar up until the point that <C> is false. It is at this point where the new program would essentially ‘break’ out of the loop and continue the program. This ‘break’ is due to the newly added goto statement which changes the flow of control by jumping to a different point in the program. A correspondence would thus be difficult to describe since, in the case of the abstract syntax tree, the lower-leveled goto statement would somehow need to show a relationship between it and the higher-leveled ‘L: alpha’ statement.

Problem 3:



Problem 4:

The best way to transform this program fragment is to assume that if the program goes back to a previous point using a condition, then that continues some outer loop. If the expression does not go back to a previous point but uses a condition, then the condition is continues the program flow.

We can first look at the loops and determine that E2 and E3 utilize a while loop to go back to the execution of alpha. Delta and eta occur if the conditions are false. Beta occurs after alpha.

loop := **true**

**while** loop

<alpha>;

<beta>;

**if** E2

**continue;**

**else**

<delta>;

**if** E3

**continue;**

**else**

<eta>

The next condition is E4, which goes to the execution of gamma which occurs after delta and before eta. Thus, an inner loop is required. E will then need to break out of this loop to continue the outer loop. Once the program is out of this inner loop, we need to check if it is due to the fact the E4 is no longer true, or if E3 is true.

loop := **true**

**while** loop

<alpha>;

<beta>;

**if** E2

**continue;**

**else**

<delta>;

exitInner := false

**repeat**

<gamma>;

**if** E3

exitInner := true**;**

**else**

<eta>;

**until** !E4 **or** exitInner

**if** exitInner

**continue;**

**else**

loop := false

Next, since E1 would skip beta, the conditional check of E2, and delta. Thus, we can let the program execute these lines if E1 is false. We can also add <nu> after the outer loop to complete the transformation.

loop := **true**

**while** loop

<alpha>;

**if** !E1

<beta>;

**if** E2

**continue;**

**else**

<delta>;

exitInner := false

**repeat**

<gamma>;

**if** E3

exitInner := true**;**

**else**

<eta>;

**until** !E4 **or** exitInner

**if** exitInner

**continue;**

**else**

loop := false

<nu>;

Problem 5:

a) Because the *or* conditional is a special case of the if-then-else conditional, we can utilize a similar evaluation format as the translate(Cond, 1) construct mentioned in class. First, we need to check the condition of the first statement. If the first statement is true, we simply evaluate the expression to true. If the first statement is false, then we recurse using translate on the next statement. If the first statement is itself another statement, we recurse using translate on that statement. If no statements after all the recursive calls evaluate to true, then the entire expression evaluates to false.

b) As previously mentioned, the *or* conditional is a special case of the if-then-else conditional which itself has a precise method of evaluation using translate(Cond, 1). One would have to, however, implement a method to determine if no condition in the expression ever evaluates to false. A possible fix to this could be to evaluate the entire expression as false if the if-then-else statement is allowed to conclude without any other statements evaluating to true first.

Problem 6:

Each subtree tree containing a single arithmetic expression would consist of 2 branches, one for each register that is either a leaf node of the abstract syntax tree, or another subtree that would determine that register through another arithmetic expression. In, any case, the number of registers required for evaluating the expression is directly proportional to the number of leaf nodes in the abstract syntax tree.