Moti Begna

CSCI 5106

Homework 3 – Rework

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. By showing that these assertions hold at certain crucial points of the program, we can use those properties to discern that the program in its entirety is doing exactly what it should be.

b) To start the proof for the correctness of this program, we have to first acknowledge the two primary loops that exist: one from line 3-15, and another from line 8-13. These loops are critical to the proof of the entire program, as they execute the core functionality of what the program is trying to do—skip over adjacent duplicates of an integer array. The first loop iterates over each new run of the integer array, while the second loop iterates over each element of the run until a new integer value is found. We can utilize Hoare's triple in order to prove that the preconditions and postconditions of these loops hold in the program.

Loop from lines 8-13:

	RAM Program	Corresponding Pascal program
P	{M[2] contains the next element in the run}	{next contains the next element in the run}
Q	Lines 8 – 13	repeat read(next); until next!=x
R	{M[2] != M[1]}	{next !=x}

Here, we see that the lines 8-13 ensures that the loop goes through various iterations of getting the next value in the integer array, and once that loop actually terminates, the value that was read in from the input is not equal to the first element of the run. Thus we have reached the next element of a new run, which leads us to the proof for the next loop.

Loop from lines 9-15:

	RAM Program	Corresponding Pascal program
P	{M[1] contains the first element of next run}	{x contains the first element of next run }
Q	Lines 9 – 15	writeln(x); repeat read(next); until next!=x; x := next
R	{M[1] contains the EOF flag}	{x == 0}

In this implementation of Hoare's triple, we see that if M[1] contains the first element of the next run (P), then the program from lines 9-15 would correctly execute, which also contain the previously mentioned loop and its instructions. Upon termination of the loop (Q), we can assume that M[1] no longer contains the next value in the array, but rather the EOF (R) flag which is signaled when the loop reaches the end of the array.

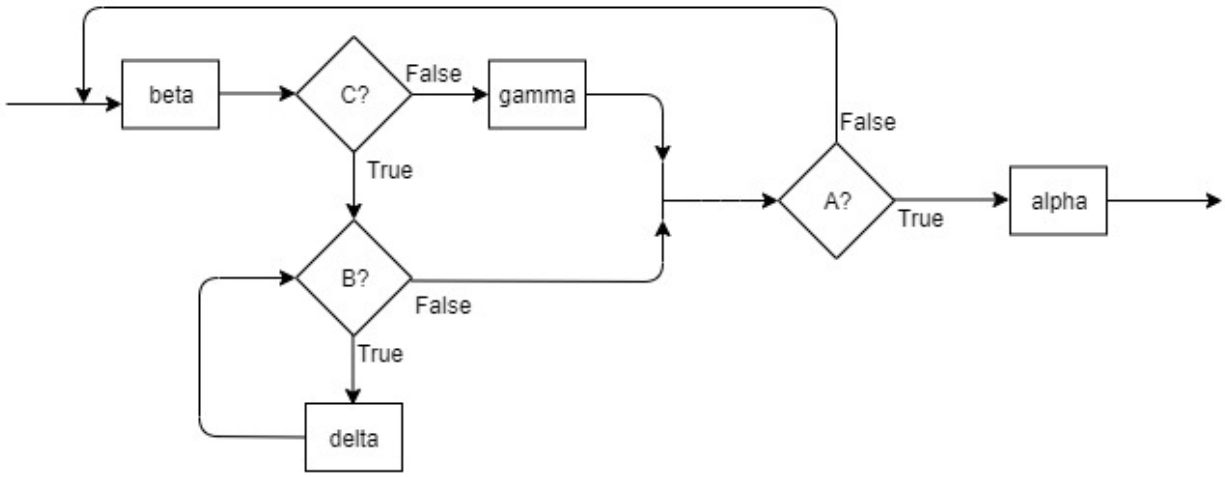
Because these essential properties of the two main loops of the program hold, we can conclude that the program correctly executes its primary functionality of removing adjacent duplicates from an integer array.

c) The first reason why the 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 comprehensible at first.

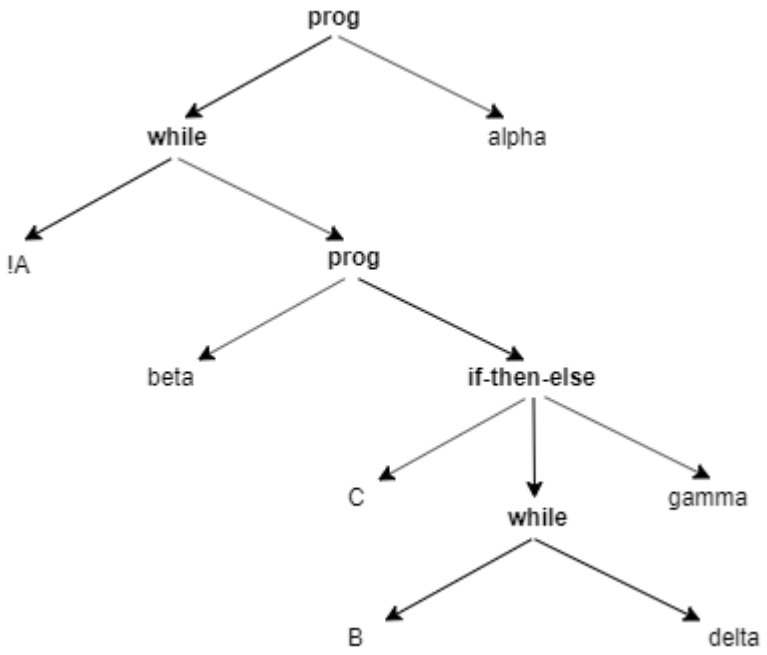
In addition to this, understanding the general program flow is difficult in the presence of goto statements. This is due to the fact that in the mindset of programing languages, programs cannot immediately forward to or backtrack from certain points of execution without the use of either some sort of loop, such as a while-do/repeat-until or the use of a conditional such as an if-then-else. This is made more complicated when multiple gotos are moving the program execution flow back and forth in a manner that makes translation into a comprehensible structure flow difficult. Because of this, proof of correctness becomes difficult since it is essential to easily determine how the control of the program gets to certain points.

Problem 2:

a)

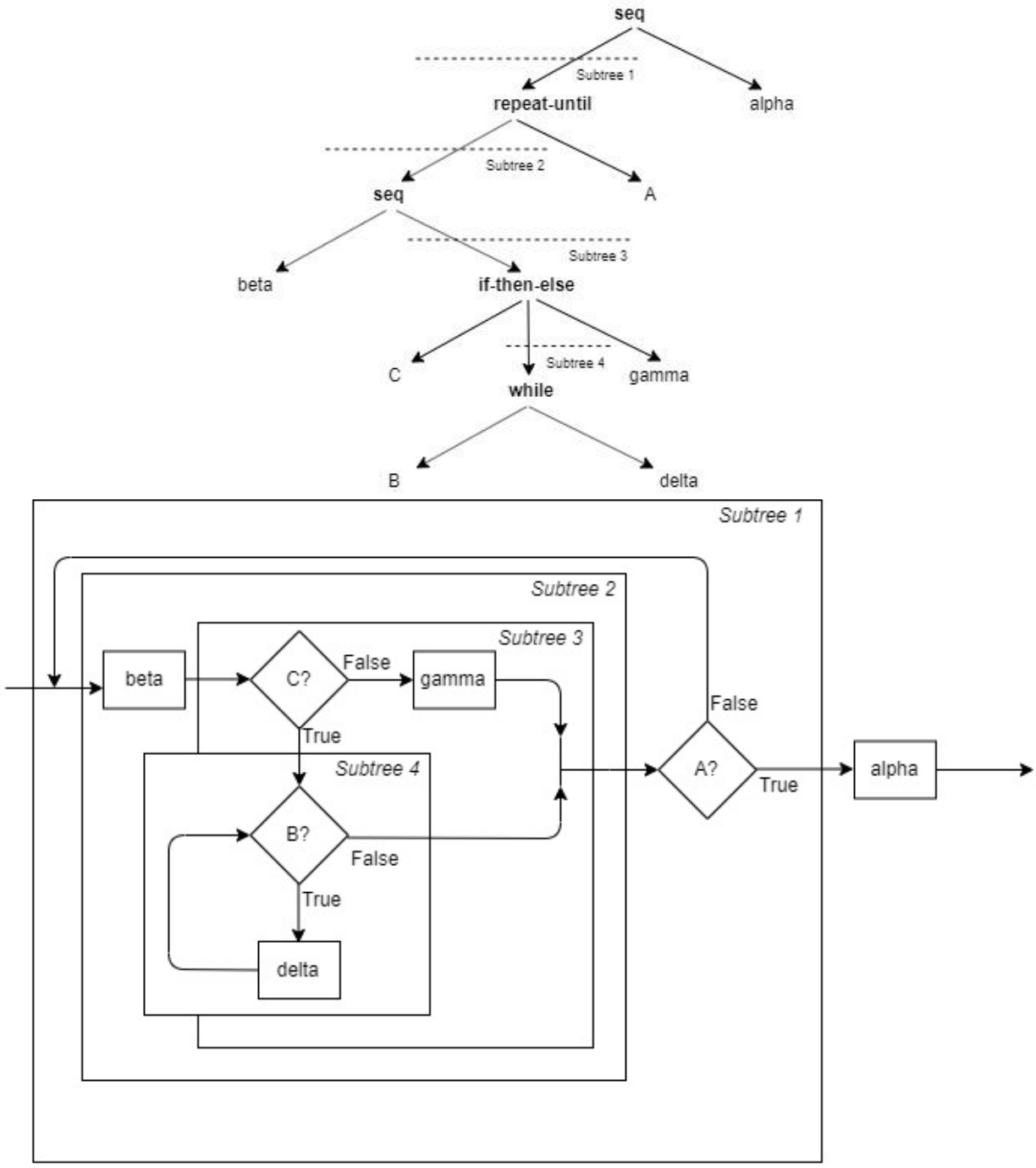


Flow Diagram



Abstract Syntax Tree

b)



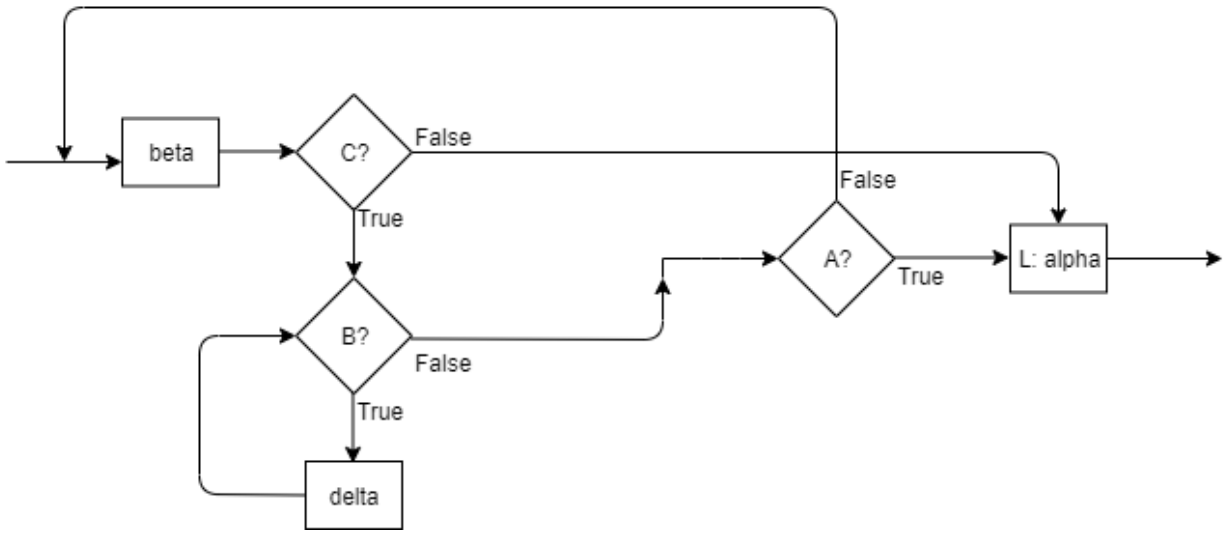
Subtree 1 : 1 edge in, 1 edge out

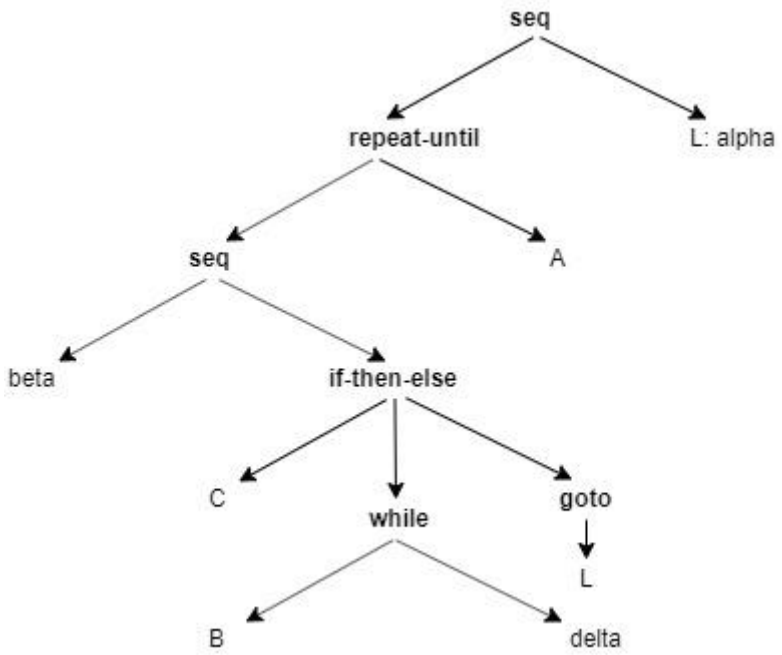
Subtree 2: 1 edge in, 1 edge out

Subtree 3: 1 edge in, 1 edge out

Subtree 4: 1 edge in, 1 edge out

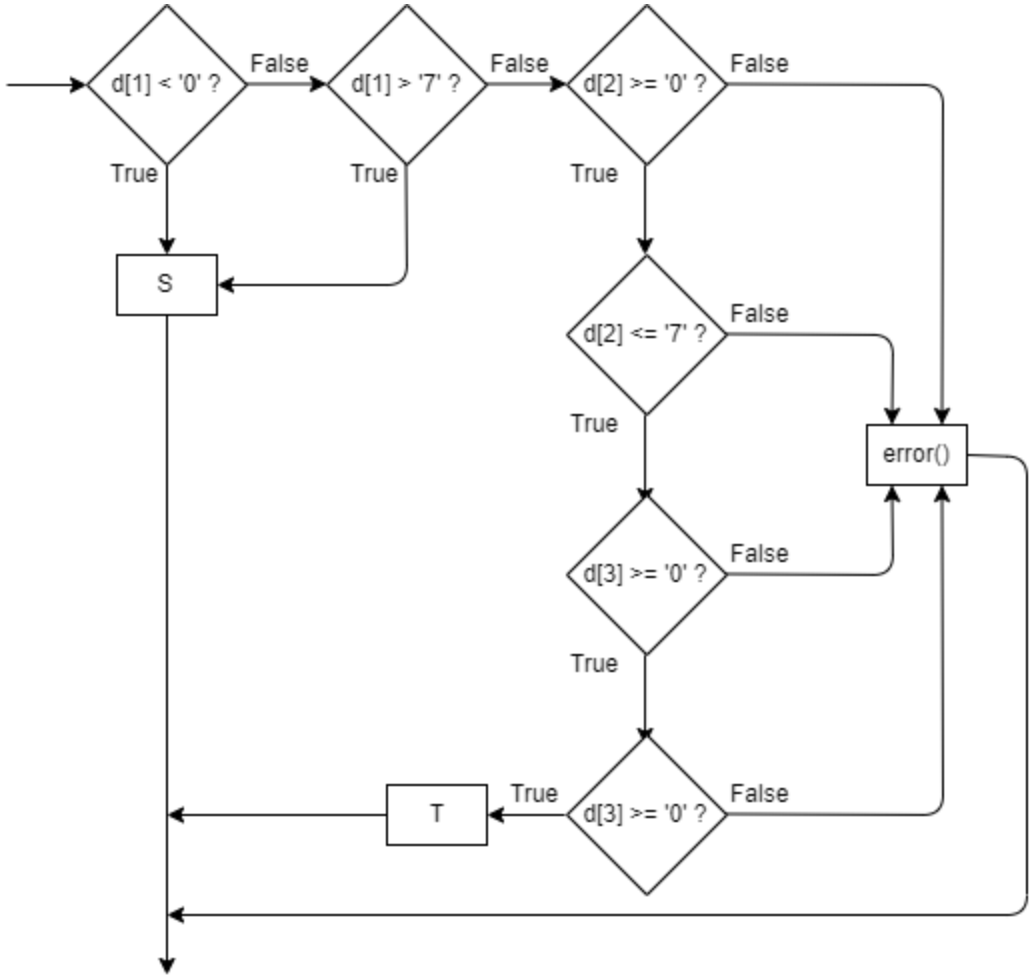
c)





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:


d[1] < '0' ?
False
d[1] > '7' ?
False
d[2] >= '0' ?
False
True
True
True
S
d[2] <= '7' ?
False
True
error()
d[3] >= '0' ?
False
True
T
True
d[3] >= '0' ?
False

Problem 4:

To transform this program fragment, we can first look at the loops and determine that E_2 and E_3 utilize some sort of loop to go back to the execution of alpha. We can utilize a repeat-until loop to gain this same affect by setting some flag that would determine whether the loop will continue or not.

```
repeat
        <alpha>;
        exit := true;
        if E1 then goto m
        <beta>;
        if E2 then
                exit := false;
        else
                <delta>;
                m: <gamma>
                if E3
                        exit := false;
                else
                        <eta>;
                        if E4 then goto m
until exit
<nu>;
```

We see that the conditional E_1 skips forward in the program flow execution if true. Thus, we can use another Boolean flag to ensure that the code prior to the execution of line m and after the conditional check of E_1 is not executed.

```
repeat
        <alpha>;
        exit := true;
        doE1 := false;
        if E1 then
                doE1 := true;
        else
                <beta>;
        if E2 and !doE1 then
                exit := false;
        else
                if !doE1 then
                        <delta>;
                else
                        m: <gamma>;
                if E3
                        exit := false;
                else
                        <eta>;
                        if E4 then goto m
until exit;
<nu>;
```

Finally, we see that the conditional E_4 returns the program execution back to line m. Thus, some sort of loop is needed in order to encapsulate the code being iterated over. We can once again use another Boolean flag to do so.

```
repeat
        <alpha>;
        exit := true;
        doE1 := false;
        if E1 then
                doE1 := true;
        else
                <beta>;
        if E2 and !doE1 then
                exit := false;
        else
                doE4 := true
                if !doE1 then
                        <delta>;
                else
                        while doE4
                                m: <gamma>;
                                if E3
                                        exit := false;
                                        doE4 := false;
                                else
                                        <eta>;
                                        if E4 then
                                                doE4 := true;
                                        else
                                                doE4 := false;
until exit;
<nu>;
```

Problem 5:

a) With this new set of rules for evaluating *or* expressions, we can see that the biggest change comes down to implementing an evaluation strategy where—in the case of an abstract syntax tree—all values at each level are being considered before the next level of values. Because of this, evaluation moves from depth-first to breadth-first where if we find any value to be T at one level, the entire expression thus evaluates to T. Thus, we recurse using translate on some arbitrary expression *only if* no value of T occurs at the same level as that expression.

b) Based on the nature of the evaluation strategy, we cannot apply a simple adaptation of a postfix-based implementation scheme. This is due to the fact that there has to be some sort of capability for the hardware to run parallel evaluations for the two parts of the *or* conditional since we are running a depth-first search method of evaluation. In addition to this, there must be some hardware mechanism that would let the process know to end computations once it as found a value of T.

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. Let *L* be the number if registers needed for the left branch and *R* the number of registers needed for the right branch, and *N* be the total needed. Assuming that the AST is recursively evaluated at each level, the evaluation of *N* would then be as follows:

1. If the left branch and the right branch are both terminals (numbers/names), then you would need 1 register to hold the evaluation of the values.
2. If the left branch is a terminal, while the right branch is a subtree, then you would need N = L + (R + 1) registers, where one register is used to remember the value of the left branch.
3. If the right branch is a terminal, while the left branch is a subtree, then you would need N = (L + 1) + R registers, where one register is used to remember the value of the right branch.
4. If both the left and right branches are subtrees, then traverse one following either rule 2 or 3.

Thus, at most, the number of *N* registers needed would be equal to the height of the abstract syntax tree, since each level traversed of the AST requires 1 register be used to remember the value of node that was not traversed from the prior level, and since registers can also be reused.