For the honors contract for my COSC314 class, I have created a program that takes a compound proposition and evaluates it, returning either true or false. The operators that the user can input to this program are NOT (~), AND (/\), OR (\/), CONDITIONAL (->), BICONDITIONAL (<->) and parentheses.

The method that I made use of to evaluate the compound propositions made use of two stacks. One stack converts the infix expression that the user has input to the program, and converts it to postfix. The second stack then evaluates that postfix expression and returns either true or false. This documentation will primarily cover the inner workings of this two-stack solution.

After the user inputs an expression, the program will call this method, evaluateExpression(). This method returns a string, which (if everything goes correctly) will be the final answer. The first step is to get the postfix expression, so the method infixToPostfix() will be immediately called.

```
public static String evaluateExpression(){

    //retrieves the postfix expression
    ArrayList<String> postfixExpression = infixToPostfix();
```

The method infixToPostfix will start by establishing an Arraylist and a Stack of Strings. The ArrayList will store the final postfix expression, and the Stack will be for processing operators in the expression, making sure to place them in the right order.

```
public static ArrayList<String> infixToPostfix(){
    ArrayList<String> postfixExpression = new ArrayList<>();

    //Stack that stores all the operators of the infix expression when looping through it.
    Stack<String> operators = new Stack<>();
```

Immediately after, there is a for each loop, looping through each value in an ArrayList of strings called “infixExpression”.

```
//loops through infix expression, evaluates each variable and operator to convert to postfix
for(String s : infixExpression){
```

“infixExpression” is essentially a parsed version of the proposition the user has input to the program. If the user inputs

“A = true”
“B = false”
“C = true”

“Proposition: a /\ (~b \/ c)”
The resulting infix expression will be: [“true”, “/\”, “(”, “~”, “false”, “\/”, “true”, “)”]

In the loop, it will check to see if the current index of the infix ArrayList is “true” or “false”. If it is, that means it has run into a variable and will immediately add it to the list without doing anything else.

```
for(String s : infixExpression){
    if(s.equals("true") || s.equals("false")){
        postfixExpression.add(s);
```

However, if it is an operator, we need to use the stack to determine where it goes. In that case, there is an inner loop that will only stop on its own when the operator stack is completely empty. The first if statement will only run in the special case if the program runs into a right parenthesis. It will pop each operator off of the stack, until it runs into a left parenthesis which it will pop and then break out of the loop.

```
}else if(validTerms.contains(s)){                                    ⚠ 15
    while(!operators.isEmpty()){
        //if we run into a right parentheses, we will want to pop everything in the stack only until we run
        //into a left parentheses.
        if(s.equals(")")){
            if(operators.peek().equals("(")){
                operators.pop();
                break;
            }
        }
        //for every other operator, we only want to check to see if the operator on top of the stack is of
        //a lower priority than it. If it is, there's no problem and we break the loop. Otherwise, we pop
        //it from the stack and try again. Also, if the operator we want to add is a left parentheses,
        //we just want to immediately break the loop without popping anything, as this parentheses is just
        //an indicator of when to stop popping the stack when we eventually run into the right parentheses.

        else if(s.equals("(") || validTerms.indexOf(operators.peek()) > validTerms.indexOf(s)){
            break;
        }

        postfixExpression.add(operators.pop());
    }
```

For any other operators, we want to break the loop if either:

1. The operator is a left parenthesis. We don’t want the left parenthesis to influence the stack at all, only act as a marker to stop popping from the stack whenever we run into the right parenthesis.
2. The operator in the stack is a lower precedence than the operator we are going to add.

This is done in the second if statement. But how does the program determine which operators take precedence over others?

When populating the validTerms list, which is primarily for checking to see if the user inputs anything that is not able to be processed by the program, the operators will be populated in order of highest precedence to lowest precedence, so higher precedence operators will have a lower index than lower precedence ones. This is what we are checking in the if statement above.

```
public static void populateValidTerms(){
    validTerms.add("~");
    validTerms.add("/\\");
    validTerms.add("\\/");
    validTerms.add("->");
    validTerms.add("<->");
    validTerms.add("(");
    validTerms.add(")");
}
```

After the inner loop concludes, the program pushes the operator to the stack, then moves to the next value of the infixExpression ArrayList.
However, if at any point in this loop the program runs into an operator that is not in the “validTerms” list, it will immediately return a null value.

```
            postfixExpression.add(operators.pop());
        }

        //We never want to push a right parentheses to the stack. When we run into one we only want to pop everything
        //from the stack until we run into a left parentheses.
        if(!s.equals(")")) {
            operators.push(s);
        }
    }else{
        return null;
    }
}
```

Finally, after the outer loop finishes looping through every element of the infixExpression Arraylist, there is another loop which loops through the operators stack and pops out every element to add to the postfixExpression. After that, we are finally able to return the postfix expression.

```
    }else{
        return null;
    }
}

//after evaluating everything, if the stack still has elements, pop each in order and add to the postfix expression
while(!operators.isEmpty()){
    postfixExpression.add(operators.pop());
}

return postfixExpression;
```

Now that the program has the postfix expression, we will go back to the evaluateExpression() method, and assign it to the variable “postfixExpression”. If it is null, an error message will be returned.

```
public static String evaluateExpression(){

    //retrieves the postfix expression
    ArrayList<String> postfixExpression = infixToPostfix();

    if(postfixExpression == null){
        return "Something went wrong. Try again.";
    }
```

A stack called “variableStack” is then created, which is where the program will store any T/F values it runs into while looping through the postfix expression.

```
//We now create a new stack to push variables (T/F values) into.
Stack<String> variableStack = new Stack<>();

//loop through the postfixExpression to start calculating the final outcome.
for(String s : postfixExpression){

    //if we run into a T/F value, push it to the variableStack.
    if(s.equals("true") || s.equals("false")){
        variableStack.push(s);
```

However, if the program runs into any operators (such as NOT, AND, OR, etc.) it will start to pop values from the variableStack, depending on how many values the operator need to work with.

For instance, NOT only applies to one variable, meaning that only one variable will be popped off the stack. Every other operator uses two, so two will be popped. A switch statement is used to check every operator and run the right method to calculate the result.

```
}else if(!variableStack.isEmpty()){
    String var1 = variableStack.pop();
    //since NOT is the only univariable operator, it only needs one variable off the top of the stack, so
    //we will check for it here.
    if(s.equals("~")){
        variableStack.push(evalNot(var1));
    }else if(!variableStack.isEmpty()){
        //For the rest of the operators, we want to check if the stack has one more element, then evaluate
        //both based on what operator it was and push it to the top of the stack.
        switch (s) {
            case "/\\":
                variableStack.push(evalAnd(var1, variableStack.pop()));
                break;
            case "\\/":
                variableStack.push(evalOr(var1, variableStack.pop()));
                break;
            case "->":
                variableStack.push(evalConditional(var1, variableStack.pop()));
                break;
```

The result is calculated by methods, which take in one or two strings (should be either “true” or “false”) and converts those values into booleans. It then does the operation with the two, then returns the result as a string. These are just two of the 5 methods used for each possible operator.

```
1 usage
public static String evalOr(String a, String b){
    boolean x = Boolean.valueOf(a);
    boolean y = Boolean.valueOf(b);
    return String.valueOf( b: x || y);
}

/**
 *  Takes one string which should always be "true" or "false", converts them into boolean values, does the not
 *  operation with it and returns the result as a string.
 *
 *  @param a The string representing a boolean variable
 *  @return String
 */
1 usage
public static String evalNot(String a){
    boolean x = Boolean.valueOf(a);
    return String.valueOf(!x);
}
```

After the result is calculated by these functions, it is pushed to the variableStack. This repeats until the loop through the entire postfixExpression ArrayList is completed. However, if the stack is empty and the loop runs into any operators at all, or the stack only has one element and the loop runs into a binary operator, it will immediately return and error message.

```
            }else{
                //If the variableStack only had one element and we ran into a binary operator, something must
                //have gone wrong
                return "Sorry, something went wrong. Expression may be malformed, please try again.";
            }
        }else{
            //If we run into an operator when the variableStack was empty, something must have gone wrong.
            return "Sorry, something went wrong. Expression may be malformed, please try again.";
        }
    }
```

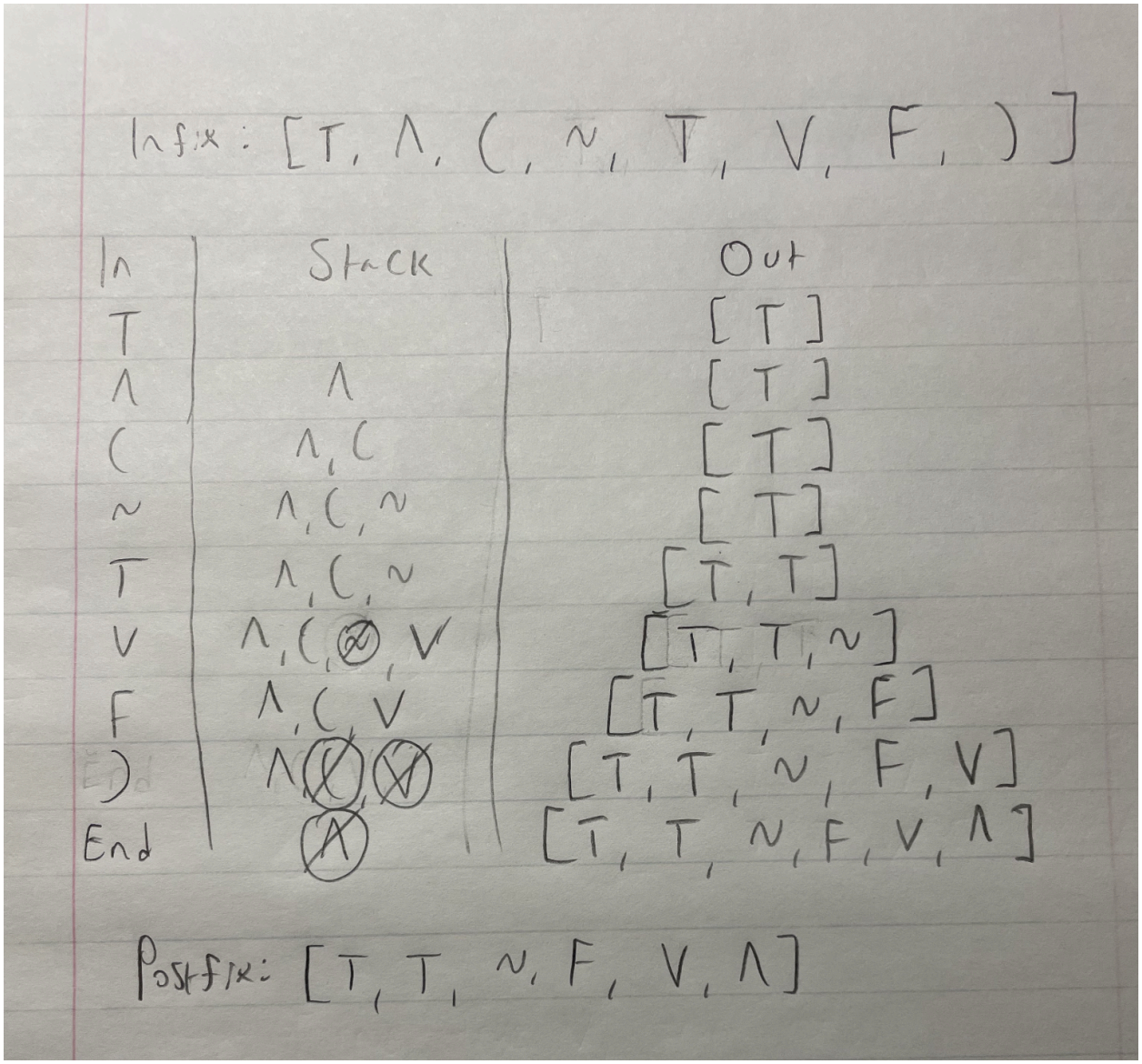
Finally, after the loop is finished, there should only be one element in the stack, which is our final answer. We return this answer, which will be printed for the user to see.

```
//If everything went well, we should have the answer as the only element in the variableStack, so we return it.
return "The value of the given logical expression is " + variableStack.pop() + ".";
```

Now for an example, take the proposition “a /\ (~b \/ c)”, where a and b are true, while c is false. We will run through the process the program goes through to solve this proposition step-by-step. First, it takes the proposition and puts it into the infixExpression ArrayList, which would look like this:

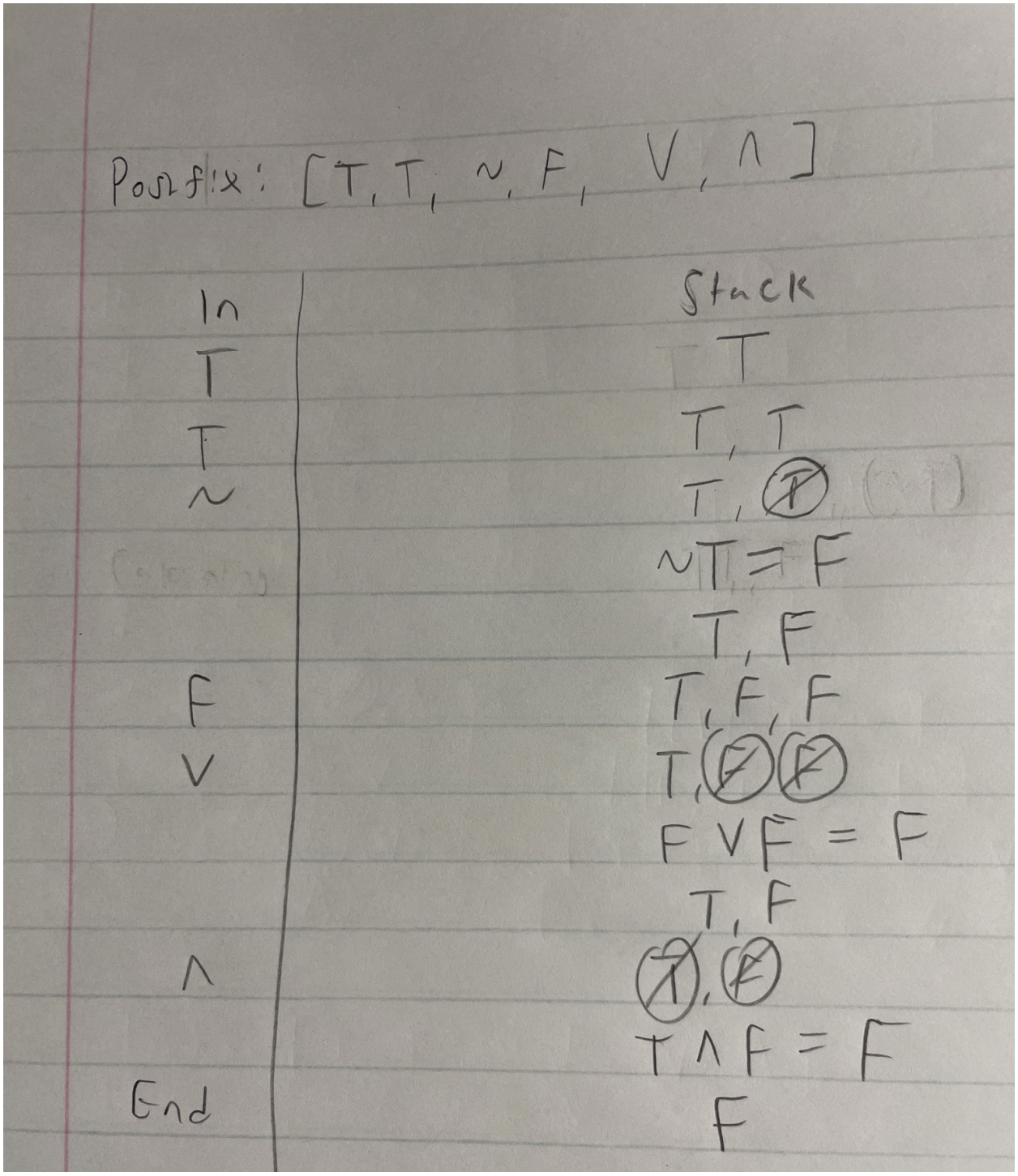
[“true”, “/\”, “(”, “true”, “\/”, “~”, “false”, “)”]

Here is a step-by-step table illustrating how the program will convert this infix list into a postfix list. The “In” column is the operand or operator currently being evaluated, the “Stack” column is the stack where the operators are pushed and popped from, and the “Out” column is the list which will contain the final postfix expression.



As you can see, if you try to push an operator to the stack, and a higher priority operator is at the top of the stack, it will get popped off into the postfix expression. An interesting thing to note is that the parentheses are never added to the postfix expression, they simply impact how the program processes the expression when converting to postfix.

So now that the program has a postfix ArrayList, we can now move on to calculating the solution.



Whenever the program runs into a true/false value, it will add it to the stack. When it runs into an operator, it pops one or two operands from the stack depending on the type of variable and does the calculation, then pushes the result back to the stack. When it is finished, there should only be one remaining true or false value, which is the answer.