Automata & Logic
Software Engineering 1

18-10-2020

Jane Buzykina
392854

Table of Contents

Introduction

This report will guide a reader through the process of completing the assignments for the ALE1 course in Fontys. It consists of 9 sections. The first five explain in detail the chosen approach as well as encountered challenges during the implementation of each of the assignments. Next three parts five an overview of the whole program, namely its Software Design, GUI, and Testing. The report finishes with the conclusions and improvements part.

Assignment 1: Parse + Tree

The goal of this phase was to make a foundation for the application, which should include main functions such as parsing the input and producing a binary tree based on it. To start with development, I first needed to decide on technologies for this project. Initially, I wanted to write a web app creating interface using React or Vue.js and backend in C#. However, due to time constraints, I decided to go with C# Win Forms, a UI framework with which I had the most experience. As a result, I could spend more time thinking about actual logic.

Having all the formal parts settled, I began to think about where to start. The first step was creating a Binary Tree data structure. The idea was the following: Binary Tree has information about its root; the root has information about its left and right child; right and left children have information about their children and so on… All the nodes were divided into types like leaf node, disjunction node, conjunction node and represented by its type class inherited from the Node abstract class. Why abstract and not interface? Simply because all its children had some implementations in common.

Now that the Binary Tree structure was there, I had to come with a way to initialize it by recursively parsing input from a user. While testing my middle versions of the parsing algorithm, I had an application breaking too many times due to wrongly formatted input. This inconvenience became a motivation to add a step of validation before parsing. During my automata course, I got familiar with regex expressions used to validate the patterns. The only problem with those was that in theory regex cannot keep track of the number of repetitions of some symbol, which was a big drawback because then parenthesis could not be counted. However, after some research online, I found an article about balancing groups that can be used in C# regex expressions. A balancing group is just like a counter, thus the number of opening and closing parentheses can be compared now. This finding helped me to come with the working validation regex. I verified its correctness by testing on few different expressions and moved on to the parsing.

Parsing was quite straightforward to me. The idea was to read input char by char, first building the left branch of the tree when it was done build the right one. Once there were no more nodes to process, the binary tree would be done. This was achieved using recursion, which acted just as a pre-order tree walk. After a lot of successful testing, this assignment was completed.

Assignment 2: Truth Table

After the lecture, all the steps of making the truth table were super clear to me. Initially, I decided to use a 2-dimensional array to store my rows and columns. However, unlikely for me later, when I tried to do the simplification assignment, I faced the problem of 2-dimensional lists getting converted to a jagged array. Thus, I had to use this data structure to avoid conversion problems.

As for the logic, it was simple:

1. Defining the size of the truth table, which was 2^n (the number of possible combinations of 0's and 1's).
2. Fill in the columns with the 0's and 1's.
3. Calculating the result based on the values in each row.

The second point from above was a bit tricky for me to implement. The idea was to calculate the number after which 0's should be changed to 1's and repeat it until all elements in the column are processed. That was achieved by checking if the current element divided by the number of repetitions had the remainder 0. The next step was to calculate the result for each row. Originally, I tried to replace the values in the leaves of the initial tree for the calculation. However, while trying that I faced an issue of a copy of the tree object being treated as a reference. After a small investigation, I understood that when you change a property of the copied object, it changes the initial object as well (pointer concept). Thus, I decided to build a new tree based on a modified formula with replaced variables by "1" or "0". It might seem that the running time of the second approach is longer, but that is not true. They are equal because recursively walking through a tree is the same as building a new tree recursively.

Lastly, the result column was read as a binary number and then converted to a hexadecimal number using the formatting methods provided by C#.

Assignment 3: Simplification

This assignment was the toughest out of all implemented so far. The requirement was to simplify the retrieved table by modifying the rows that resulted in 1. Unfortunately, I understood the simplification method wrong from the beginning, and as a result, my implementation was far from what was needed. The algorithm constructed can be summarized as follows:

1. Identify rows resulting in "1".
2. Compare every row to every other row.
3. Count the number of symbols the rows have in common.
4. If only one symbol differs, substitute it with '*' in both rows.
5. If no rows were modified - exit, else go to step 2.

Some of the produced new rows were too simplified, while some of the others were not simplified enough. To solve these two problems, I had to understand their cause. After some debugging, I reached the following conclusions:

1. The first error occurred because there was no check if the row can be simplified further.

2. The second error was there because the rows were modified before a single iteration of the checks finished. Hence, the comparison would show that rows have more than one different symbol, while in fact, it was only one.

To fix these issues, I adjust the initial algorithm to have versioning control of the rows and to make a check if the row can be simplified further. The versioning control was done through a copy of initial rows, which got changed every time the check was passed. Thus, the next iteration was using the result rows of the previous. As for a check, I verified if the row I want to simplify would interfere with one of the rows that resulted in "0". If no, the change was accepted. As the last step, I removed the duplication rows using LINQ query methods.

For the validation, I used both solutions for some propositions in the presentation slides and the website, which can generate the simplified truth table.

Assignment 4: Normalization

For this assignment, I needed to provide disjunctive normal forms for both original and simplified truth tables. From the slides, I got a clear idea of how to construct the dnf. For every row resulting in "1", I used a loop to look through each row in the table. Inside that loop, there was another that was looking through each column value in the current row. When the column value was "0", it would be substituted with the negation of the table variable in the same position. When it was "1", the original value of that variable would be added. Then conjunction operator was placed in between every resulted variable of that row and disjunction in between every computed row.

Moreover, the dnf for simplified was done with the same logic, except for row values that resulted in "*", which were skipped due to "*" being neither "1" nor "0".

Assignment 5: Nandify

This was the easiest assignment out of all. To convert an operator into NAND, I used recursion. While going through every node, I replaced its operators with equivalent NAND version. After the new formula with only NANDs could be created, I modified the regex to also accept this type of expressions. Also, I modified the algorithm of building the proposition tree which included now "*" operator too. Thus, NAND expressions were successfully parsed and generated.

Software Design

GUI

Testing

Conclusions and Improvements

This course is in my personal top of the Fontys courses. It was as challenging as enjoyable. During the course, I had a chance not only to learn more, but also to apply the knowledge I got previously from my premaster courses. Implementing it, gave me a deeper understanding of the learned theory. The course upgraded my logical thinking skills and will be a nice addition to my CV.

Overall, I am proud of the created solution. However, the existing solution still can be improved. The running time should be optimized in order to process more complex expressions. Additionally, I would modify existing GUI. As for me, it looks too old-fashioned. Also, I feel like there can be more time spend on unit testing.