# CPSC-402 Report

Stephen White

May 17, 2022

**Abstract**

In short, the goal is to search for all occurrences of a small string within a larger string. We are trying to emulate the text searching abilities of Regular Expressions.

## Contents



# 1 Introduction

My name is Stephen White. I am a Computer Science major with a minor in Business Administration at Chapman University.

# 2 Homework

This section contains the solutions to the homework.

## 2.1 Week 1

### 2.1.1 Homework 1.1

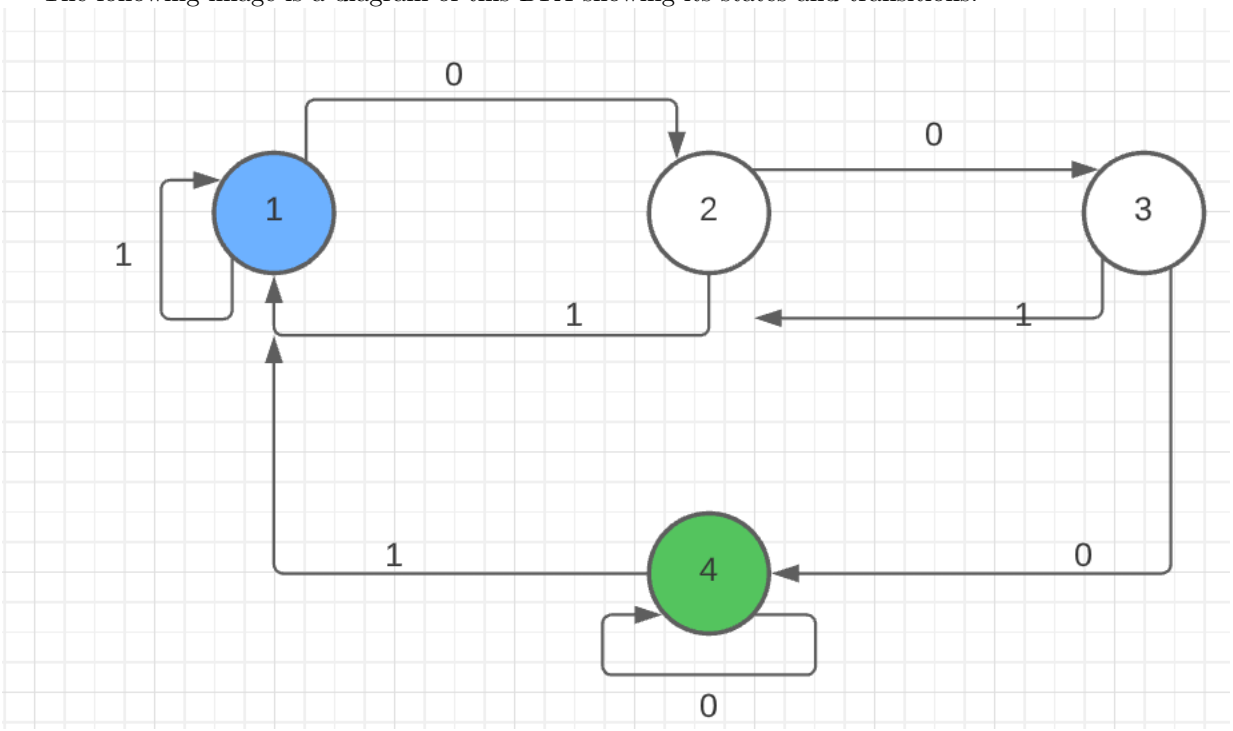
The homework discussed in this section can be seen here. Part 1 of the homework is exercise 2.2.4 item B. The problem can be found on page 54 of the textbook (page 69 of the pdf).

The first question is as follows: "Give DFA's accepting the following languages over the alphabet {0,1}: The set of all strings with three consecutive 0's (not necessarily at the end)."

My answer to this question is as follows:
Since we know that this DFA's alphabet can only consist of 0's and 1's, and we are searching for three consecutive 0's, looking like this: 000, we can formally defined the DFA as

$$\{\omega \mid \omega \; is \; of \; the \; form \; x000y \; for \; some \; strings \; x \; and \; y \; consisting \; of \; 0's \; and \; 1's \; only\}$$

The following image is a diagram of this DFA showing its states and transitions.



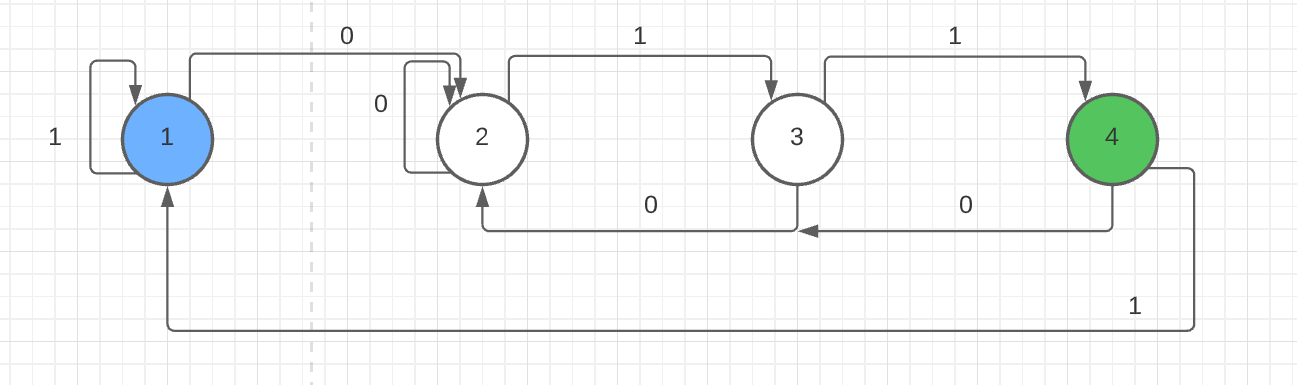

(I have chosen blue to signify the initial state and green to signify the success state)

The next question asks "Give DFA's accepting the following languages over the alphabet {0,1}: The set of strings with 011 as a substring."

My answer to this question is as follows:
Since we know that this DFA's alphabet can only consist of 0's and 1's, and we are searching for the string 011, we can formally defined the DFA as

$$\{\omega \mid \omega \; is \; of \; the \; form \; x011y \; for \; some \; strings \; x \; and \; y \; consisting \; of \; 0's \; and \; 1's \; only\}$$

The following image is a diagram of this DFA showing its states and transitions.




(I have chosen blue to signify the initial state and green to signify the success state)

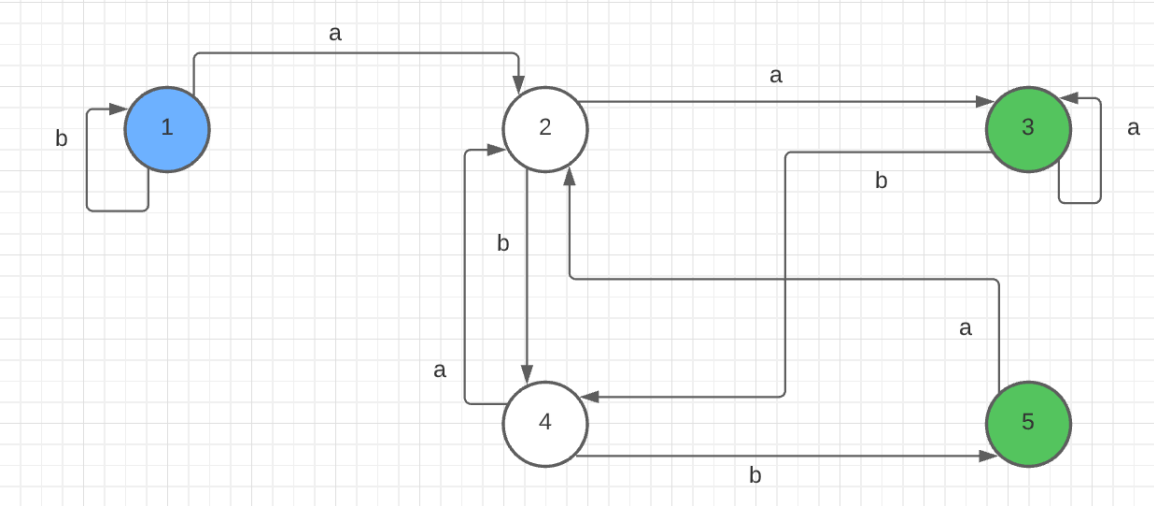
### 2.1.2 Homework 1.2

The homework discussed in this section can be seen here.

Part 1 of this assignment is to answer the following question:

"Does the file contain the sequence aa or abb? For example, if the file is aaabbaabb, the output should be [2,3,5,7,9]. The challenge here is to not write a program that runs twice through the file, once to search for aa and once for abb, but to find a way of running through the file only once and to simultaneously search for both strings."

I chose to write the program in C# because I am most familiar with it, and I believe it has a good mix between control and programming speed. That code can be found here on my GitHub. To aid me in my understanding of the problem, I made the following diagram of the DFA I was creating:




Part 2 of the homework was to complete exercise 2.2.10 (the proof being optional) on page 54 of the textbook (page 70 of the pdf).
The question reads as follows:

| | 0 | 1 |
|---|---|---|
| $\rightarrow A$ | $A$ | $B$ |
| $*B$ | $B$ | $A$ |

"Informally describe the language accepted by this DFA."

I tackled this problem by first breaking down the chart, to understand what I was being asked. The chart is similar to a Punnett Square. The left column is a list of the possible current states (an input). The top row is a list of possible character inputs. The rest of the chart is the next state (the output) determined by that row's current state and current character. Since the only possible characters are 1 and 0, the language accepted by this DFA is a set of strings or words created by any combination of 0's and 1's.

## 2.2 Week 2

### 2.2.1 Homework 2.1.1 Composing Automata

The homework discussed in this section can be found here.
**Exercise 2.3.4**
Part 1 of this assignment is to answer the following questions:

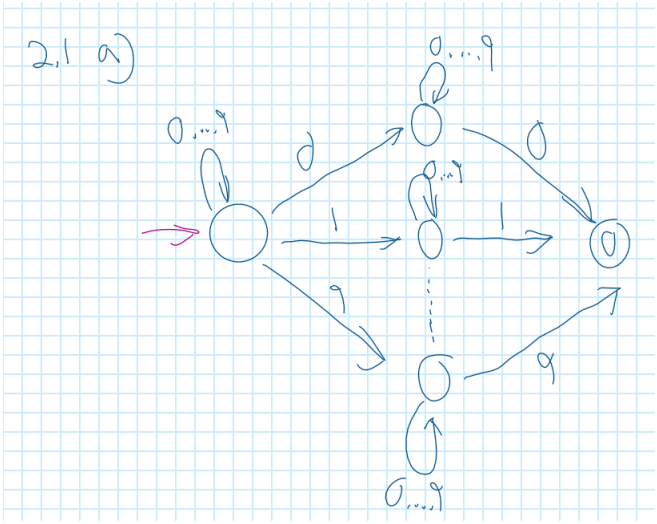
Find NFAs that recognize:

a) The set of strings over alphabet $\{0, 1, \ldots, 9\}$ such that the final digit has appeared before.

b) The set of strings over alphabet $\{0, 1, \ldots, 9\}$ such that the final digit has *not* appeared before.

c) The set of strings of 0's and 1's such that there are two 0's separated by a number of positions that is a multiple of 4. Note that 0 is an allowable multiple of 4.

**Part A:**
The NFA I created is outlined in the diagram below.




Included in my diagram is the following definition for some of the notation.

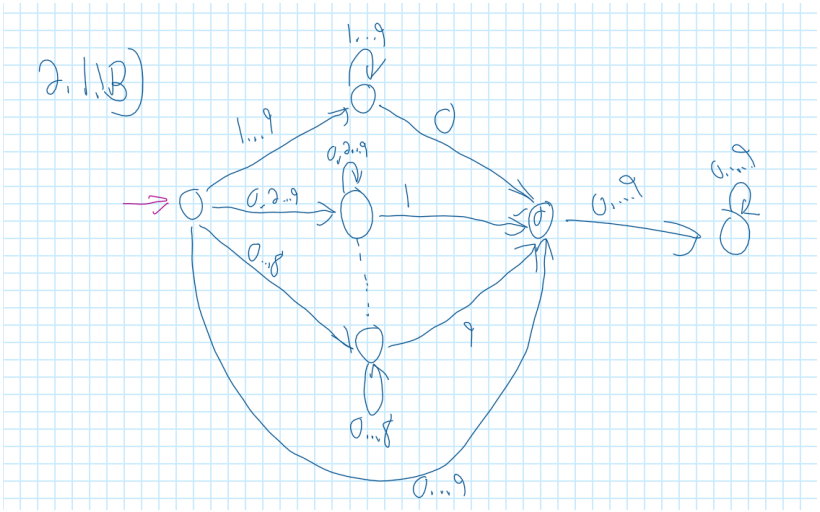
Where $\Sigma := \{0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9\}$

The regular expression for part A of exercise 2.3.4 is as follows:

*( 0 + ... 9) * ( 0 ( 0 + ... 9 ) * 0 + ... )*

(We had gone over this in class.) The "..." is used to simplify and shorten the writing of the expression rather than writing all the digits 0 through 9.
I used the "..." as a way to simplify the diagram and make it more readable by not writing every state, but enough to understand the diagram. The idea of the NFA is to ensure that the only way to reach the success state is to have crossed over the last digit before. The most helpful part for my understanding was to look at the transition to the success state has the same condition as the transition to the mid-state. This ensures that the digit has been seen before in the string.

**Part B:**
The NFA I created is outlined in the diagram below.



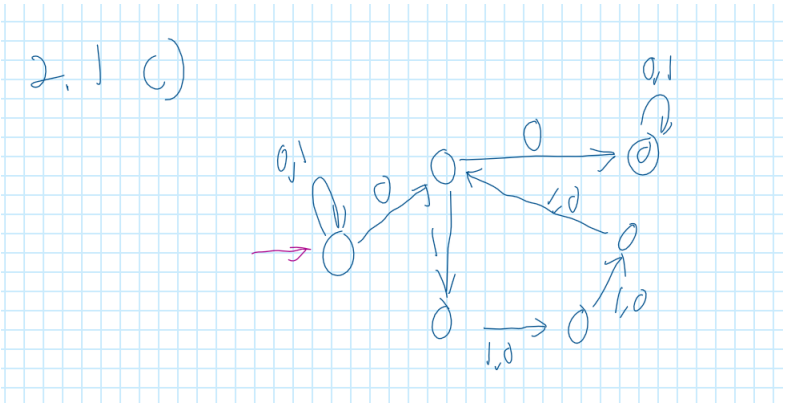
Where $\{1 - 9\} := 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9$
$\epsilon ::= \{1 - 9\} * 0 + \{0, 2 - 8\} * 1 + ...\{0 - 8\} * 9$

The regular expression for part B of exercise 2.3.4 is as follows:

*( 1 + ... 9 ) * 0 + ( 0 + 2 + ...9 ) * 1 + ( 0 + 1 + 3 + ...9 ) * 2 + .... (0 + ...8 ) * 9 + $\varepsilon$*

This regular expression gets all the strings where the last digit has no appeared before in the string.
The main idea of this NFA is to ensure that the only way to reach the success state is to have never seen the last digit before in the string. The part that helped me understand this NFA the most is knowing that it is effectively the complement of part A. So to accomplish this, I used the same foundation as the one from part A, but the transitions from the initial state are all digits except the one I'm looking for. For example, the top path is only successful if the final digit is zero, so the first transition includes all numbers 1 through 9, to prevent the possibility of 0 being seen elsewhere in the string. In addition to this, I added another state after the success to prevent the success state from firing if the transition is taken before the end of the string.

**Part C:**
The NFA I created is outlined in the diagram below.




The regular expression for part C of exercise 2.3.4 is as follows:

$$(0+1)^*\ 0\ ((0+1)^4)^*\ 0\ (0+1)^* \tag{1}$$

For this problem, I needed some help. I utilized the website Regex101 to help me create this regular expression. Since the above expression is not directly applicable in most programming languages, the regular expression I came up with in Regex is this:" *.* 0 ((0|1){4})* 0 .* ".
The main challenge with this problem is to count multiples of 4, including 0. I tackled this challenge by having 3 states with 4 transitions that effectively "count" the digits between 0's. Since the problem states that the entire string doesn't have to be encased by the 0's, I have loops on the initial and success states that enable the NFA to count an entire word that contains the 0's.

**Exercise 2.5.3**
Part 2 of this assignment is to answer the following questions:

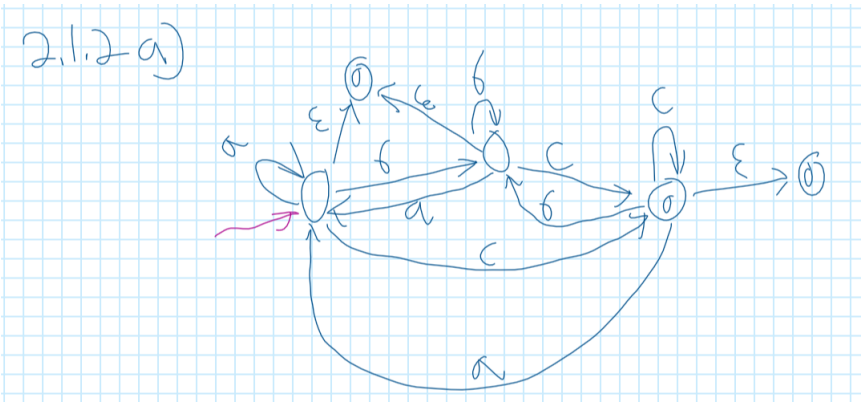
Find ε-NFAs that recognize:

a) The set of strings consisting of zero or more $a$'s followed by zero or more $b$'s, followed by zero or more $c$'s.

b) The set of strings that consist of either 01 repeated one or more times or 010 repeated one or more times.

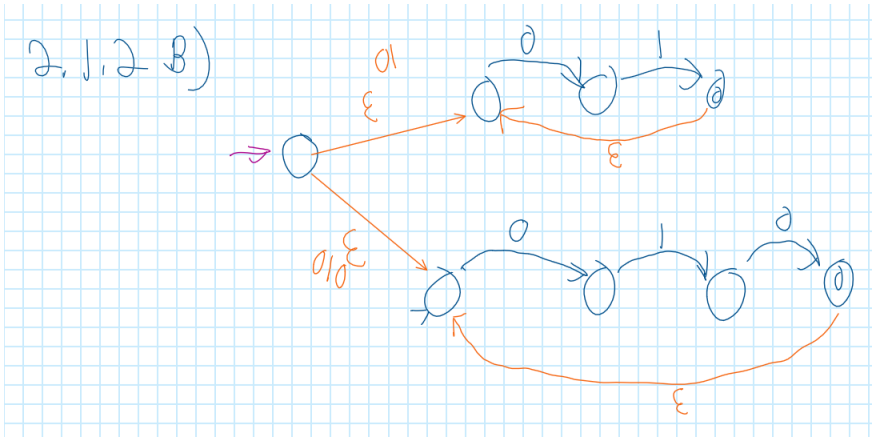
c) The set of strings of 0's and 1's such that at least one of the last ten positions is a 1.

**Part A:**
The NFA that I created is outlined in the diagram below.



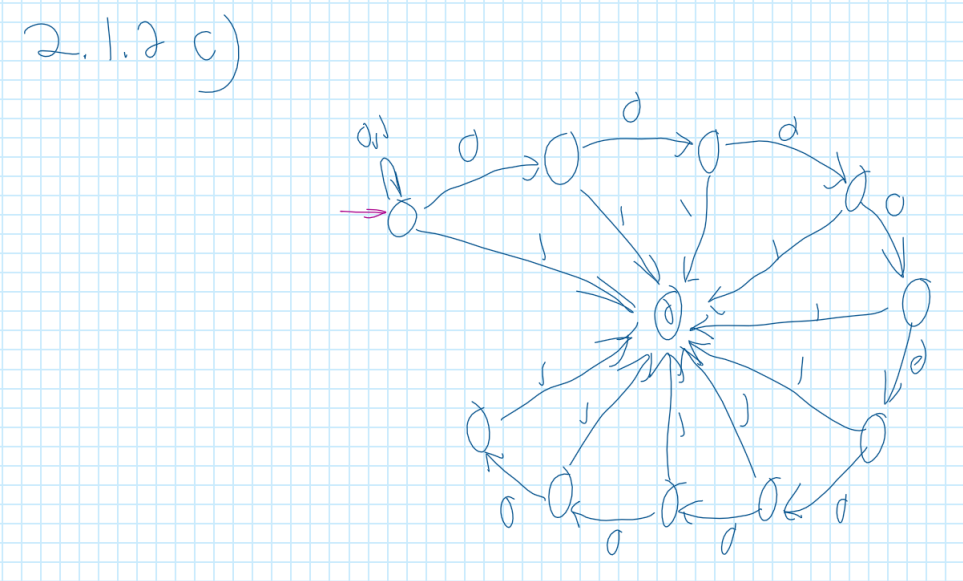

The main challenge with this NFA is the fact that there can be zero of each character. This diagram gets complex very quickly because we are looking for a pattern of alphabetical strings, but there is no guarantee that it will contain an 'a', 'b', or 'c'. While solving this problem, there was some confusion around the phrase "consisting of." I chose to take that similar to "having substrings of" rather than "being entirely made of." If it meant "a string entirely made of a's, b's, and c's in alphabetical order", then the diagram would be slightly different and less complex.

**Part B:**
The NFA that I created is outlined in the diagram below.




This problem had "composition" written all over it. From the start, I noticed that this would be two automata working in parallel. The top section is what identifies the "01" repeating pattern while the bottom section identifies the "010" pattern. To "join" these two sections, I performed epsilon jumps to the beginning of each section immediately from the initial state. To simplify the possibility of a repeating pattern, I simply jump from the success states back to that section's starting state.

**Part C:**
The NFA that I created is outlined in the diagram below.




The main challenge with this problem was figuring out how to count 10 positions from the last. If I were implementing this program, I would simply iterate from the back of the string rather than the start, but we are unable to do that with our current setup. To overcome this, I created a loop of both 0's and 1's on the initial state to allow for strings longer than 10. I then have 9 states (10 transitions) that upon finding a 1, enter the success state. If one of those states finds a 0, it will simply move onto the next position.

**Exercise 2.5.3**

**Part A:**

The regular expression for part A is as follows:

$$a^*\ b^*\ c^*$$

This was was very simple because the * is what allows the preceding character to appear anywhere between 0 and unlimited number of times.

**Part B:**

The regular expression for part B is as follows:

$$(01)(01)^* + (010)(010)^*$$

For this problem, I needed some help again. The regular expression I came up with in Regex101 is this: *(01)(01)+ | (010)(010)+*

**Part C:**

The regular expression for part C is as follows:

$$(0+1)^*1(0+1)^n\ ,\ where\ 0 <= n <= 9 \tag{2}$$

Since this question says that the number 1 must appear in the last 10 digits and we have the ability to use exponents in our definition, I'm using N as a way to ensure that the 1 can show up anywhere in the last 10 positions. I also had some help from Regex101 for this problem and the equivalent expression I got in Regex101 while I was working on this problem was " *.* 1 (1|0){0,9}$* ".

## 2.3 Week 3

### 2.3.1 Homework 3.1 Converting NFAs to DFAs

The homework discussed in this section can be found here.

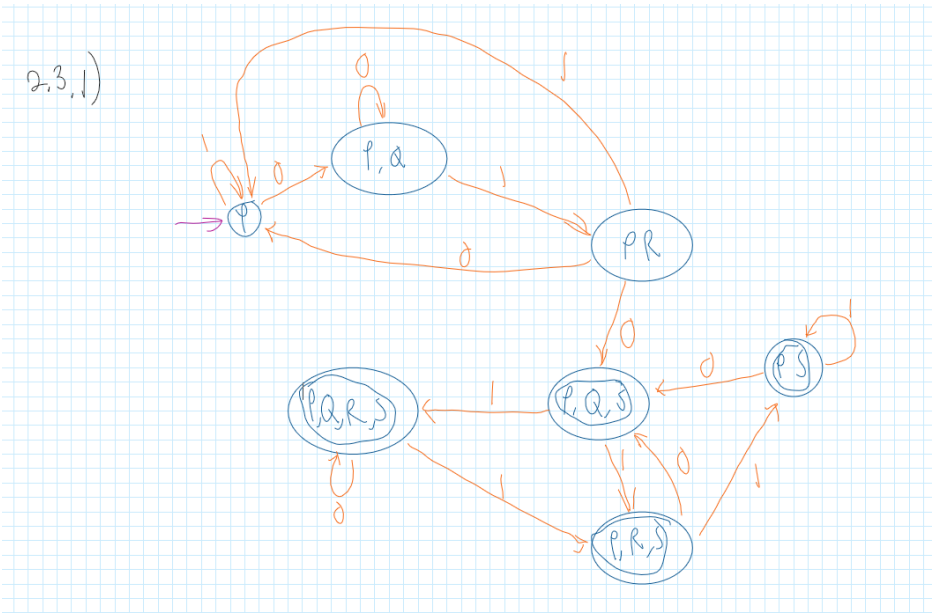
**Exercise 2.3.1**

Part 1 of this assignment is to answer the following question:

**Exercise 2.3.1 :** Convert to a DFA the following NFA:

| | 0 | 1 |
|---|---|---|
| $\to p$ | $\{p, q\}$ | $\{p\}$ |
| $q$ | $\{r\}$ | $\{r\}$ |
| $r$ | $\{s\}$ | $\emptyset$ |
| $*s$ | $\{s\}$ | $\{s\}$ |

The diagram for the DFA can be found below:

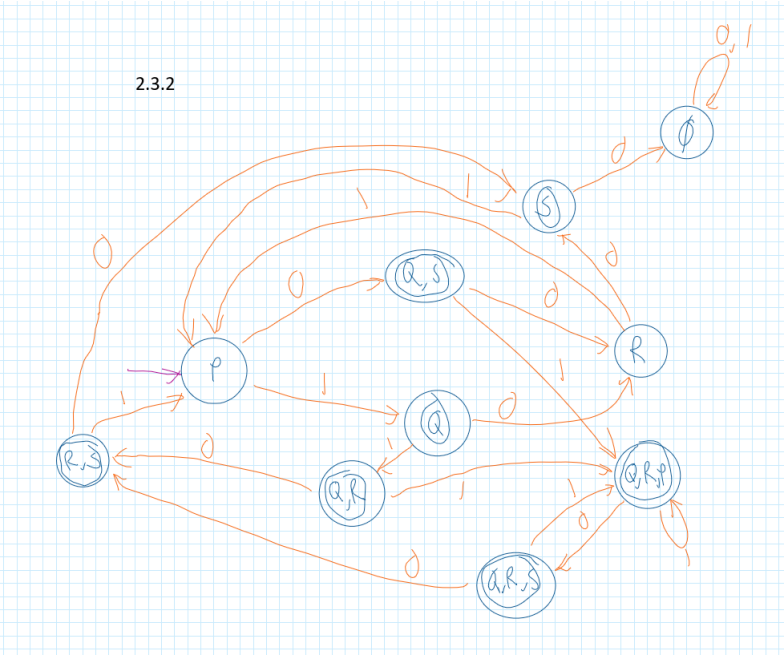



**Exercise 2.3.2** Part 2 of this assignment is to answer the following question:

**Exercise 2.3.2 :** Convert to a DFA the following NFA:

| | 0 | 1 |
|---|---|---|
| $\to p$ | $\{q, s\}$ | $\{q\}$ |
| $*q$ | $\{r\}$ | $\{q, r\}$ |
| $r$ | $\{s\}$ | $\{p\}$ |
| $*s$ | $\emptyset$ | $\{p\}$ |

The diagram for the DFA can be found below:




**Exercise Converting NFAs to DFAs in Haskell**

Part 2 of this assignment is to complete the following task:

"Put the definitions of dfa_initial, dfa_final, dfa_delta in the report and explain your code in a paragraph. Also put your modified version of automata05.hs in your github repository.)

The file automata05 provides a template, in which you have to modify lines 71, 72, 73. Complete the definition of nfa2dfa as described in the three bullet points above and test it with various examples."

My code can be seen below:

---

```
-- convert an NFA to a DFA
nfa2dfa :: NFA s -> DFA [s]
nfa2dfa nfa = DFA {
  -- exercise: correct the next three definitions
  dfa_initial = [nfa_initial nfa],
  --dfa_final = let final qs = True in final,
  dfa_final =
   let
    --final var1 var2 = (var1 var2)
    --final nfa = disjunction nfa
    --final
    --final [s] = disjunction(nfa_final s)
        --final (q:qs) = final qs
      --final var1 = (final (nfa_final var1))
      --final qs = [nfa]
      final qs = map (disjunction) (qs)



   in final,
  dfa_delta =
   let
    delta qs c = qs
```

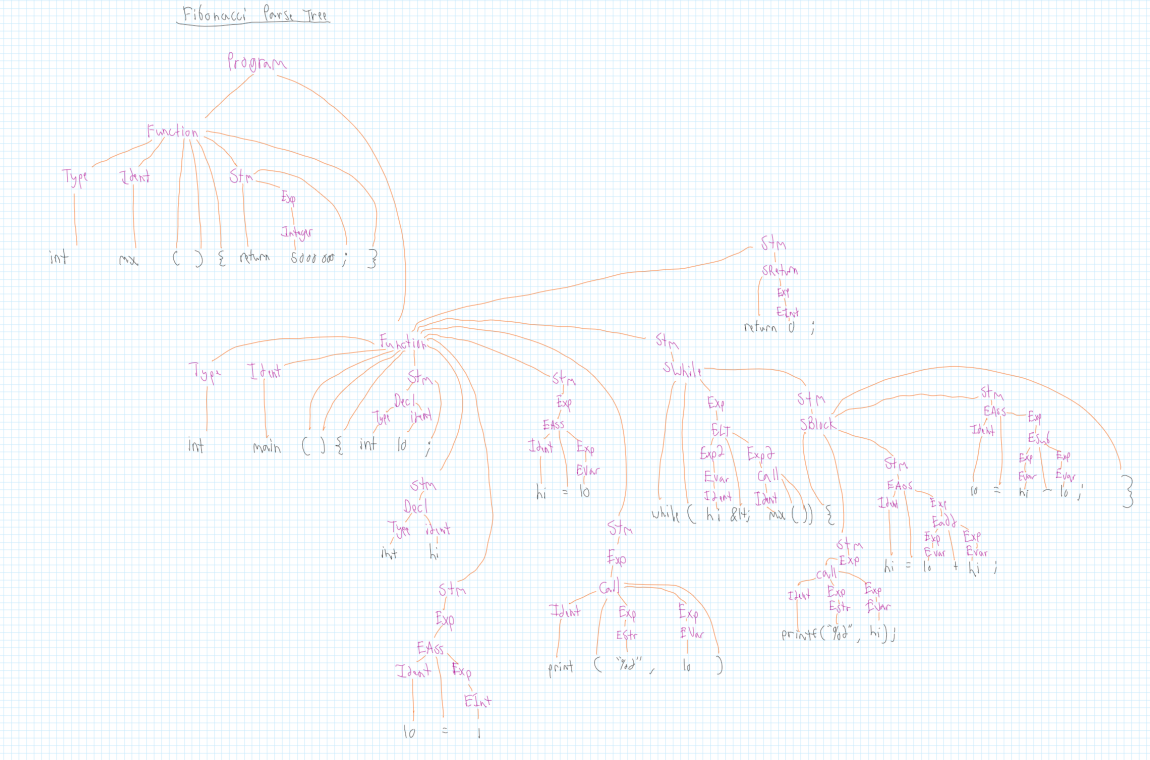
```
  --delta [] c =
  --delta (q:qs) c =
in delta }
```

Sadly, I was unable to complete this section. I am confident in my answer for dfa_initial, but I'm far from confident for dfa_final and dfa_delta. I tried to comment all of my attempts in order to show the many thought processes I had. I also hope that one of them was close. I think the hardest part for me was understanding the data types of all of the variables and knowing what was being passed into the function in the first place. I was really struggling with understanding how to use what felt like Object Oriented Programming inside of a functional language.

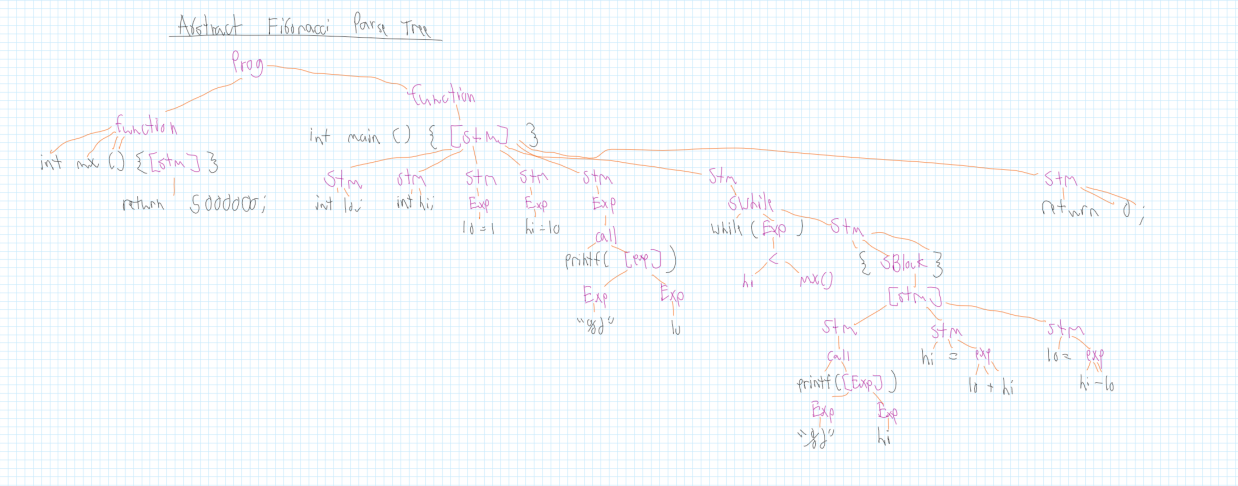
## 2.4 Week 4

The homework for week 4 can be found here.
Both parts of this assignment involved the code for C–, and that can be found here.

The point of the homework was to create concrete and abstract syntax trees for the basic program in C–. My Concrete Syntax Tree can be found below. To get a better look, here is a link to the photo on GitHub.




My Abstract Syntax Tree can be found below. To get a better look, here is a link to the photo on GitHub.




# 3 Project

## 3.1 Initial Project Description

For the project, I will be using C as my programming language and I will be using the GCC compiler to compile it to Assembly. My C program will simply be a hard coded number and printing out the factorial of that number. My C program can be found here. My Assembly can be found here.

```c
#include <stdio.h>

//Created a program does factorial
int main()
{
    int num = 3;

    int product = 1;

    while(num > 0)
    {
        product *= num;

        num = num - 1;
    }

    printf("Product:",product);

    return 0;
}
```

The following is the GCC assembly Output.

```
.file "CProgram.c" --> This line says where the program came from
  .text --> This tells what encoding the file is in
  .globl  main --> This says that the "main" pointer is global.
  .type main, @function --> This says that the "main" pointer points to a function
main: --> this is the label for the beginning of the main function
```

```
.LFB0: --> this is the label for the opening bracket in the main function
 .cfi_startproc --> used in debugging. Initializes internal data structures allowing the
     function to be in the .eh_frame
 endbr64 --> inserts a NOP into the pipeline to prevent data hazards or corruption.
 pushq %rbp --> this pushes a new context onto the stack because we entered a new code
     block.
 .cfi_def_cfa_offset 16 --> Changes the absolute offset that will be added to a defined
     register to compute the CFA address
 .cfi_offset 6, -16 --> the previous value of register 6 is saved at the offset -16 from
     the CFA pointer.
 movq %rsp, %rbp --> memory copies the base point value into stack pointer reference,
     giving it a reference for where it is in memory.
 .cfi_def_cfa_register 6 --> will now use register 6 instead of the old CFA register.
     Offset is unchanged.
 subq $16, %rsp --> integer subtraction. Moving the current pointer to show that we are in
     a new statement block.
 movl $3, -4(%rbp) --> initializing the value of 3 into a new "num" variable. ("num" =
     -4(%rbp))
 movl $1, -8(%rbp) --> initializing the value of 1 into a new "product" variable.
     ("product" = -8(%rbp))
 jmp  .L2 --> an unconditional jump to label L2
.L3: --> label L3. This is the inside of the while loop.
 movl -8(%rbp), %eax --> this is putting the value of product into the register EAX as a
     temporary variable.
 imull -4(%rbp), %eax --> Integer Multiplication. this is multiplying num by the temporary
     value in the register.
 movl %eax, -8(%rbp) --> this is copying the new value of (product' * num) into the
     register for product.
 subl $1, -4(%rbp) --> integer subtraction. Subtracting 1 from num in place.
.L2: --> label L2
 cmpl $0, -4(%rbp) --> This is the condition in the while loop. Checks if "num" is greater
     than 0.
 jg .L3 --> conditional jump when "num" > 0 is true. If true, it will enter the while.
 --> if we reached here, we are outside the while loop.
 movl -8(%rbp), %eax --> this is putting the value of product into the register EAX as a
     temporary variable.
 cltq   --> converting integer into a Long for printing.
 movq %rax, %rdi --> Puts the literal string "Product" onto the stack so it can be an
     argument in the printf function.
 movl $0, %eax --> EAX is expected to hold the number of vector registers for the printf
     arguments, but we are simply printing an integer, so there is no need for a vector
     register.
 call printf@PLT --> calls the printf function.
 movl $0, %eax --> puts a 0 into the EAX register for the return statement.
 leave --> leaves the current statement block
 .cfi_def_cfa 7, 8 --> changing offset for the CFA register 7.
 ret --> returning from the subroutine.
 .cfi_endproc --> The ending to the .cfi_startproc to signify the end of the function to
     the debugger.
.LFE0: --> this is the section that helps the system handle exceptions and adds information
    for the debugger.
 .size main, .-main
 .ident  "GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0"
 .section .note.GNU-stack,"",@progbits
 .section .note.gnu.property,"a"
 .align 8
```

```
 .long 1f - 0f
 .long 4f - 1f
 .long 5
0:
 .string  "GNU"
1:
 .align 8
 .long 0xc0000002
 .long 3f - 2f
2:
 .long 0x3
3:
 .align 8
4:
```

---

# 4 Homework Continued

## 4.1 Week 10

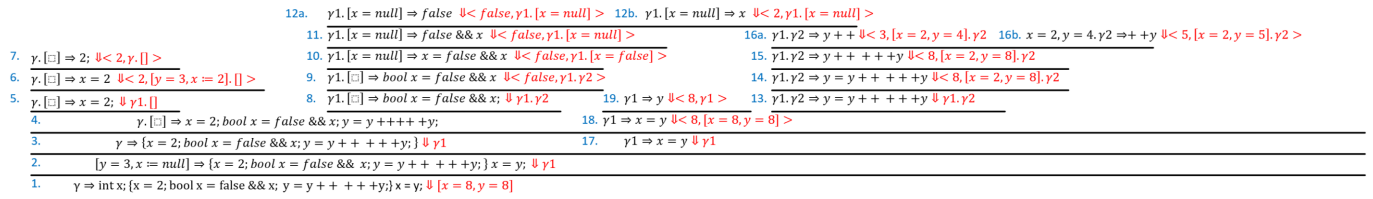
### 4.1.1 Creating a Proof Tree

The instructions for this assignment can be seen here. The point of this assignment was to create the tree of steps the interpreter would take while completing the following program:

---

```
int x;
{
    x = 2 ;
    bool x = false && x ;
    y = y++ + ++y ;
}
x = y ;
```

---



$$1.\quad \frac{}{\gamma \to Tx; \downarrow \gamma(x := null)}$$

$$2.\quad \frac{\gamma \to s \downarrow \gamma' \qquad \gamma' \to S2...Sn \downarrow \gamma''}{\gamma \to S1...Sn \downarrow \gamma''}$$

$$3.\quad \frac{\gamma. \to S1...Sn \downarrow \gamma'.\delta}{\gamma \to \{S1...Sn\} \downarrow \gamma'}$$

$$4.\quad \frac{\gamma \to s \downarrow \gamma' \qquad \gamma' \to S2...Sn \downarrow \gamma''}{\gamma \to S1...Sn \downarrow \gamma''}$$

5. $$\frac{\gamma \to e \downarrow < v, \gamma` >}{\gamma \to e; \downarrow \gamma`}$$

6. $$\frac{\gamma \to e \downarrow < v, \gamma` >}{\gamma \to x = e \downarrow < v, \gamma`(x := v) >}$$

7. $$\frac{}{\gamma \to x \downarrow v}$$

8. $$\frac{\gamma \to e \downarrow < v, \gamma` >}{\gamma \to e; \downarrow \gamma`}$$

9. $$\frac{}{\gamma \to Tx; \downarrow \gamma(x := null)}$$

10. $$\frac{\gamma \to e \downarrow < v, \gamma` >}{\gamma \to x = e \downarrow < v, \gamma`(x := v) >}$$

11. $$\frac{\gamma \to a \downarrow u \qquad \gamma \to b \downarrow v}{\gamma \to a \&\& b \downarrow u \ x \ v}$$

12. $$\frac{}{\gamma \to x \downarrow v}$$

13. $$\frac{\gamma \to e \downarrow < v, \gamma` >}{\gamma \to e; \downarrow \gamma`}$$

14. $$\frac{\gamma \to e \downarrow < v, \gamma` >}{\gamma \to x = e \downarrow < v, \gamma`(x := v) >}$$

15. $$\frac{\gamma \to x \downarrow v \qquad \gamma \to x \downarrow v}{\gamma \to a + b \downarrow < v, \gamma` >}$$

16. $$\frac{}{\gamma \to x + + \downarrow < v, \gamma`(x := v + 1) >}$$

17. $$\frac{\gamma \to e \downarrow < v, \gamma` >}{\gamma \to e; \downarrow \gamma`}$$

18. $$\frac{\gamma \to e \downarrow < v, \gamma` >}{\gamma \to x = e \downarrow < v, \gamma`(x := v) >}$$

19. $$\frac{}{\gamma \to x \downarrow v}$$

# 5 Conclusions

To make a poor analogy, this course is like a construction worker learning physics. Technically, a builder can put a building together and follow the instructions on the blueprints without knowing why the engineer did any of it. The problem with the builder working like that is he has no way of utilizing his building skills without someone telling him what to build. This is analogous to the life of a software engineer because yes, technically any 12-year-old can create a basic video game in python by following a YouTube tutorial, but there is so much more to programming than following instructions.

This course allows us to go from basically utilizing the tools provided to us to being able to create our own tools to better tackle the next problem we face. In my life this has been made very clear through talking to my dad about the problems he faces on the day to day. My dad is a UCLA graduate with a degree

in computer science and he said that Compiler Construction is the most useful class he's taken. In his work, he created a parser to automatically extract information from and categorize his employer's mess of custom stock trade orders. As I took this class, we were able to talk more and more about how his parser works and why he does what he does. I am able to understand the problems he faces and how I might be able to tackle the same solutions.

This class was also incredibly effective at teaching me how to use the basic tools to create more complex ones. This class hammers recursion into our brains. Recursion has always been an incredibly challenging topic for me. Challenging to create. Challenging to debug. Challenging to read. This class has taught me how to approach recursion and the kinds of problems that can best be overcome using recursion.

This class is a fantastic class to follow up from taking Computer Architecture my previous semester. Over the years I've been learning about how to create software and how to build solutions, but I never understood how a computer actually worked. I finally took Computer Architecture and then I was still left with questions. Thankfully less questions, but still questions. The biggest one was "so I can build software solutions and I know how we turned a chunk of silicon into a fancy math machine... now how does my code get to the assembly for the computer?" This class has now filled in the whole stack. From keyboard stroke to transistor, I have an (admittedly a general and high level) understanding of every step in between.

I think this class has allowed me to better understand the strange quirks and mishaps that occur when programming. I am far more knowledgeable on why some syntax just doesn't work. I am far more capable of predicting what certain code will do before I run it, because I now (at least mostly) understand how it is actually being computed. As a software engineer, I am now able to better understand the tools at my disposal, and I am able to better apply myself to any problem. I've graduated from the builder to the engineer, the programmer to the software engineer.

## References


[] *assembly - GAS: Explanation of .cfi_def_cfa_offset*. en. URL: https://stackoverflow.com/questions/7534420/gas-explanation-of-cfi-def-cfa-offset (visited on 05/16/2022).

[] *CFI directives (Using as)*. URL: https://sourceware.org/binutils/docs/as/CFI-directives.html (visited on 05/16/2022).

[God] Matt Godbolt. *Compiler Explorer*. en. URL: https://godbolt.org/ (visited on 05/16/2022).

[] *Guide to x86 Assembly*. URL: https://www.cs.virginia.edu/~evans/cs216/guides/x86.html (visited on 05/16/2022).

[Joh] Hopcroft John E. *Introduction to automata theory, languages, and computation*. 3rd. original-date: 2007. HMU. URL: http://ce.sharif.edu/courses/94-95/1/ce414-2/resources/root/Text%20Books/Automata/John%20E.%20Hopcroft,%20Rajeev%20Motwani,%20Jeffrey%20D.%20Ullman-Introduction%20to%20Automata%20Theory,%20Languages,%20and%20Computations-Prentice%20Hall%20(2006).pdf.

[] *macos - Why is %eax zeroed before a call to printf?* en. URL: https://stackoverflow.com/questions/6212665/why-is-eax-zeroed-before-a-call-to-printf (visited on 05/16/2022).

[] *x86 - assembly cltq and movslq difference*. en. URL: https://stackoverflow.com/questions/37743476/assembly-cltq-and-movslq-difference (visited on 05/16/2022).

[] *x86 - What does cltq do in assembly?* en. URL: https://stackoverflow.com/questions/6555094/what-does-cltq-do-in-assembly (visited on 05/16/2022).


[] [] [God] [Joh] [] [] [] []