Language and Logic
Assignment #1

24/04/2018

Helena Tevar

Problem 1

Design an NFA for the following, then convert it to a the corresponding regular expression:

1. All Hexadecimal numbers divisible by 6 with a remainder of 1 or 5.
2. All strings over {a,b,c} with at least two occurrences of abc and an odd number of as.
3. All strings over {x,y,z} where x occurs an odd number of times or y and z occur an even number of times.
4. $L = w \in \{0, 1, 2\}^*$ where a string of exactly 5 symbols contains at least two 1s or at least one 2(but not both).

Solution Part One
In order to create a NFA of hexadecimal number divisible by 6 with a remainder of 1 or 5 we first get 6 subsets of numbers divided by their remider.

$$\begin{aligned} 6n &= \{0, 6, C\} \\ 6n+1 &= \{1, 7, D\} \\ 6n+2 &= \{2, 8, E\} \\ 6n+3 &= \{3, 9, F\} \\ 6n+4 &= \{4, A\} \\ 6n+5 &= \{5, B\} \end{aligned}$$

Using a transition table in Table 1 with this subsets and naming the states of our automata as the remainders, I got the subsets that would reach an accepting state q_1 and q_5. The final solution of an NFA may be seen in Figure 1. In order to make the automata clearer, the subsets of numbers in base of their reminder would have a name, starting with A = {0, 6, C}, B = {1, 7, D}, C = {2, 8, E}, D = {3, 9, F}, E = {4, A} and F={5, B}.
As a small addition, the Figure 10 includes a funny graph I found by myself while looking for inspiration and is added at the end of the assignment.

	$A = \{6n\}$	$B = \{6n+1\}$	$C = \{6n+2\}$	$D = \{6n+3\}$	$E = \{6n+4\}$	$F = \{6n+5\}$
q_0	q_0	q_1^*	q_2	q_3	q_4	q_5^*
q_1	q_4	q_5^*	q_0	q_1^*	q_2	q_3
q_2	q_2	q_3	q_4	q_5^*	q_0	q_1^*
q_3	q_0	q_1^*	q_2	q_3	q_4	q_5^*
q_4	q_4	q_5^*	q_0	q_1^*	q_2	q_3
q_5	q_2	q_3	q_4	q_5^*	q_0	q_1^*

Table 1: Transition Table for Hexadecimal numbers multiple of six

Problem 1 continued on next page...

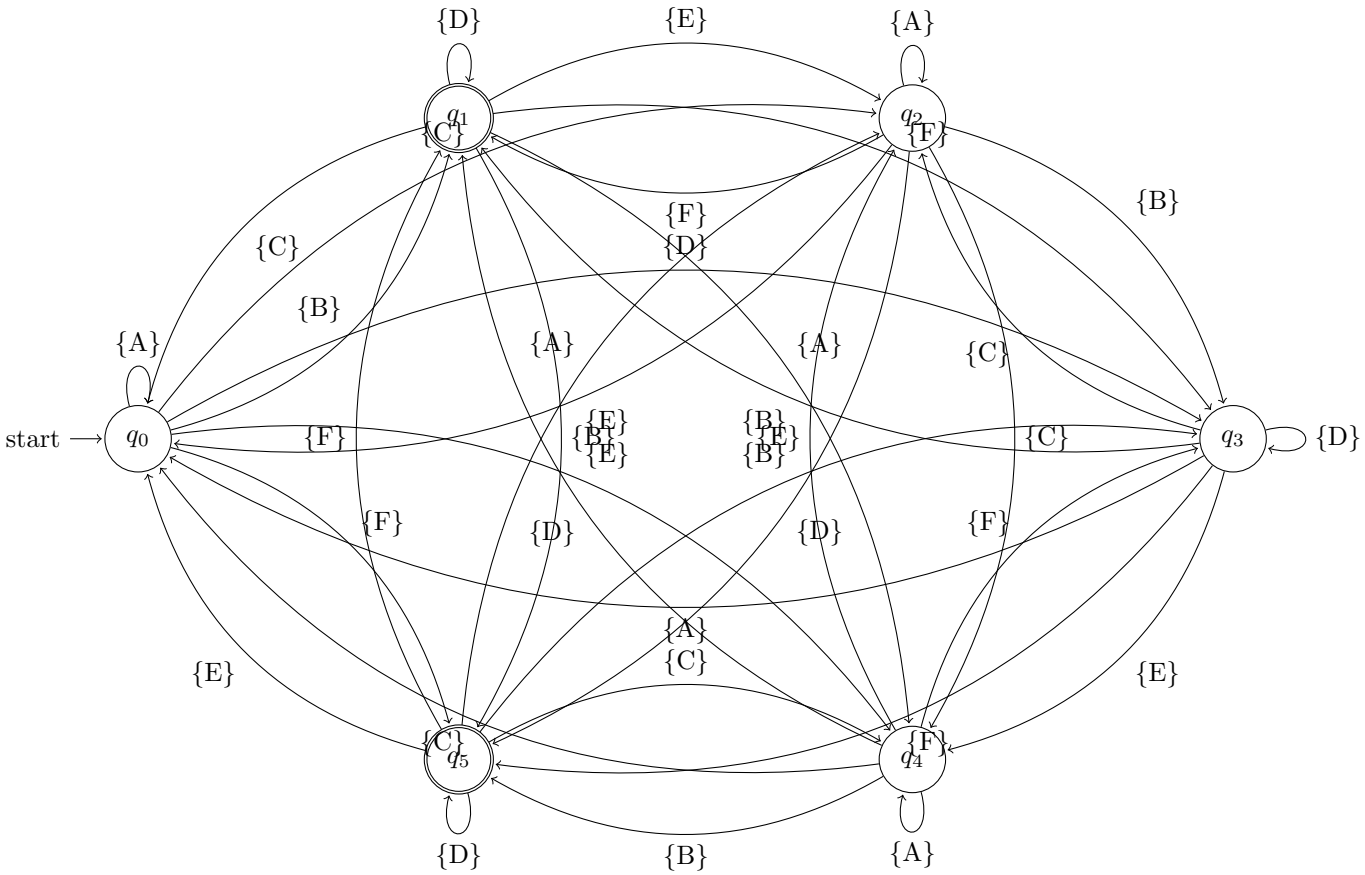

Figure 1: Hexadecimal NFA

Solution Part Two

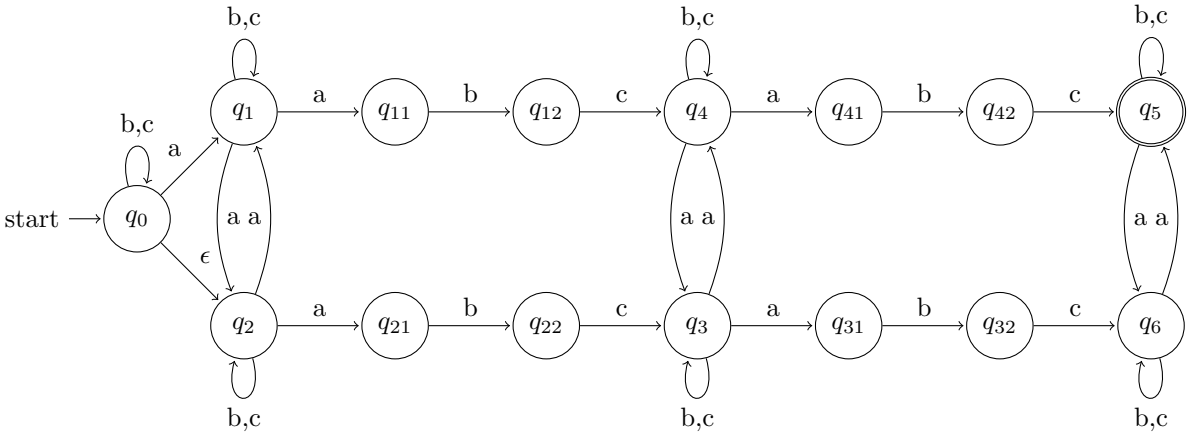

Figure 2: NFA Ex: 1.2

This exercise needed to take care of the difference of odd and even a's in the string. In order to do it, I used odd states for those strings that had odd numbers of a's and even states for strings with a even number of a's to make the automata in Figure 2 clear to read and understand. For instance, if a string has 3 a's, it would

move around the automata through an odd state q_1 for the first "a", q_2 for the second "a" and q_3 state for the last "a". This solution focus on the typical algorithm to find odd numbers $2n+1$. Our automata has to have two groups of *abc* characters, so we have the $2n$ part already solved and we don't care how many even numbers of a's we put between the groups, because while they are even, they will be multiples of 2. We only need to fix the +1 part, that may be in different places between the *abc* subsets, for instance: At the beginning, in the middle, at the end and in between all the spaces of the subset. If we name the subset $E = \{even\ a's\}, O = \{odd\ a's\}$ the possibilities of getting an odd number of a's is the next:

$$(E^*abc\ (O\ abc\ E^*|\ E^*abc\ O)|O\ abc\ (E^*abc\ E^*|\ O\ abc\ O))$$

To create a regular expression, based in the order before, is easy and is only needed to transform the even and odd subsets into its regular expressions. For $E = \{((b|c)^*a(b|c)^*a(b|c)^*)^*\}$ and for Odd numbers, E+1 being $O = \{((b|c)^*a(b|c)^*a(b|c)^*)^*a(b|c)^*\}$

$(((b|c)^*a(b|c)^*a(b|c)^*)^*abc(((b|c)^*a(b|c)^*a(b|c)^*)^*a(b|c)^*abc((b|c)^*a(b|c)^*a(b|c)^*)^*$
$|(b|c)^*a(b|c)^*a(b|c)^*)^*abc((b|c)^*a(b|c)^*a(b|c)^*)^*a(b|c)^*)|((b|c)^*a(b|c)^*a(b|c)^*)^*a(b|c)^*abc($
$((b|c)^*a(b|c)^*a(b|c)^*)^*abc((b|c)^*a(b|c)^*a(b|c)^*)^*|((b|c)^*a(b|c)^*a(b|c)^*)^*a(b|c)^*abc((b|c)^*a(b|c)^*a(b|c)^*)^*a(b|c)^*))$

Part Three

In order to a better understanding of this exercise, I created a truth table, showed in Table 2, to have a clear order how this automata should work when x is odd or y,z are even. Following the truth table from

x	y,z	OR
odd	odd	true
odd	even	true
even	even	true
even	odd	false

Table 2: Truth Table Exercise 1.3

Table 2 we see that the automata should check for all odd x's and even yz's. There is no need to check all the combinations of both. The fist idea I had was simply accepting strings with odd x's and yz even with a typical "counter", for instance $\overrightarrow{Start}\ q_0 \rightarrow q_1^* \leftrightarrow q_2$ for odds and $\overrightarrow{Start}\ q_0^* \leftrightarrow q_1$ for even, the problem was when a string as $yxxx$ entered and I should check all the characters, so I decided using the possibilities of an NFA to solve it. If we have a string as $yxxx$, the first character would go through a "y and z counter" but at the same time it will go to an "x counter" directly to the even x's state, although we do not know how many x's there are at this point, we do know that zero x's are even. Next characters are three x's, so the automata will fail when going through the "yz counter" but pass through the "x counter". This also happens with the "z", if you have zero of them, then is even for the automata. The automata shown in Figure 3 works as presented for all strings with odd number of x and even number of y and z.
To get a regular expression, is just needed to follow the truth table in Table 2:

$$(x^{2n+1}y^*z^*|x^*y^{2n+1}z^{2k+1})$$

Problem 1 continued on next page...

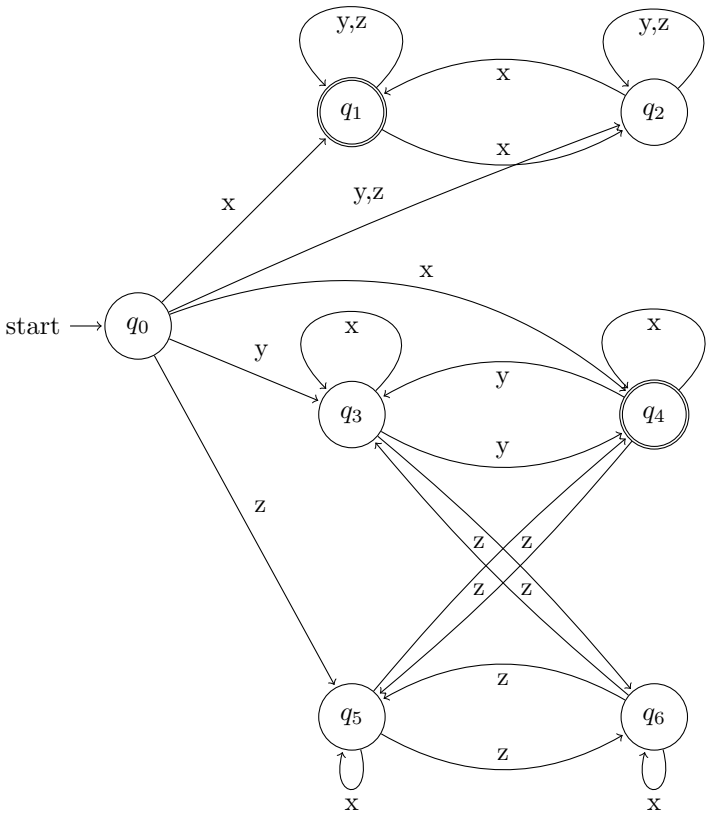

Figure 3: Solution 1.3

Part Four

After testing a lot of different ideas for automatas, that will mostly go through all the possibilities of 5 characters I got the idea that we do not need the position of the numbers in the strings, this automata does not need to work with permutations but with combinations, and with that in mind, the automata goes from absurdly big to something clearer (but still big). I created an automata that will work in the same way with $0, 1, 1, 0, 0$ than with $0, 1, 0, 0, 1$. The different states of the automata would be referring the number on 1 and 2 we have in a specific position rather that all the possibilities.
For further explanation:

- All states starting with the number 1 refer to strings with only zeros untill that position.
- All states starting with the number 2 refer to strings with only one 1 and no 2 untill that position.
- All states starting with the number 3 refer to strings with at least two 1:s and no 2 untill that position.
- All states starting with the number 4 refer to strings with at least one 2 and no 1:s untill that position.
- All states starting with the number 5 refer to strings with one only 1 and at least one 2 untill that position.
- All other states would have at least two 1:s and at least one 2, and therefore wouldl not be accepted.
- The second cypher in every state points at the position of the last consumed character in the string.

For instance, q_{34} would refer to strings that would have accumulated at least two 1:s and no 2 untill the fourth character, i.e., 0110, 1110 or 1111 . The automata accepts all the strings that end up in states q_{35}, q_{45}, q_{55}, as is showed in Figure 4.

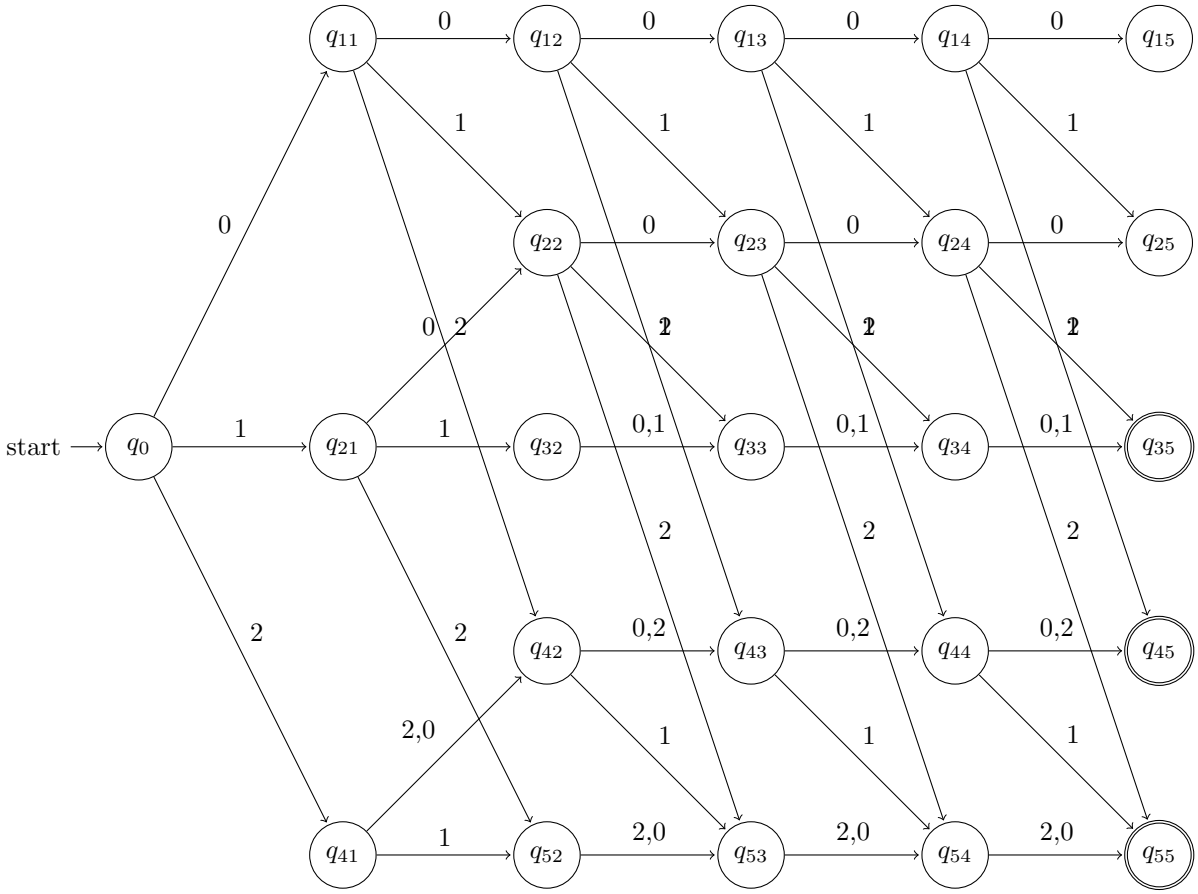

Figure 4: Exercise 1.4 Solution

Problem 2

Convert the following NFAs to a DFAs.

Solution One

In order to convert an NFA with e-closure to a DFA, the first step would be to create a transition table that will facilitate the understanding of the transitions for our future DFA.

	A	B	ϵ
q_1	q_2	$\emptyset$	q_3
q_2	$\emptyset$	q_3	$\emptyset$
q_3	$\emptyset$	q_4	$\emptyset$
q_4	q_3, q_5	$\emptyset$	$\emptyset$
q_5	$\emptyset$	$\emptyset$	q_2

Table 3: Transition Table First NFA

Using the transitions between the different nodes, we can recognize the sets of states that can be reached with only ϵ moves. The epsilon closure sets from the NFA are:

$$E(1) = \{q_1, q_3\}$$
$$E(2) = \{q_2\}$$
$$E(3) = \{q_3\}$$
$$E(4) = \{q_4\}$$
$$E(5) = \{q_5, q_2\}$$

We will use the ϵ-closure sets in a new transition table. When creating a transition table with ϵ-closure, it may be possible to find new states. When a new state is found, we will take it as current state and find its transitions. This step will be repeated until no new states are found.

	A	B
E(1) q^*_{13}	q_2	q_4
E(2) q_2	$\emptyset$	q_3
E(3) q_3	$\emptyset$	q_4
E(4) q_4	q_{35}	$\emptyset$
E(5) q^*_{52}	$\emptyset$	q_3
q^*_{35}	$\emptyset$	q_{43}
q^*_{43}	q_{35}	q_4

Table 4: Transition Table First NFA with ϵ-closure states

After all the information gathered about the new ϵ-closure states in Table 3, we can use it to create a DFA as seen in Figure 5. We will take as initial node q_{13} and all the nodes that include at least one accepting state would share this, so the accepting states would be: q_{13},q_3, q_{52} and q_{35}.

An important mention to do, as seen in the transition table with ϵ-closure states shows, some of the final states are unreachable. For example, the initial state should be also an accepting state, but is not reachable by any other state. State q_{52} is also unreachable, and by that, unnecessary and may be removed.

Some states have empty sets as result of the no existence of transitions, for example q_2 has no transition with the character "a". In order to make all the characters reachable, we will add escape nodes in replacement for the empty sets, for instance q_6.

The final automata would be create taking into consideration all the mentions before, as seen in Figure 5.

Problem 2 continued on next page...

	A	B
E(1) q_{13}	q_2	q_4
E(2) q_2	q_6	q_3
E(3) q_3^*	q_6	q_4
E(4) q_4	q_{35}	q_6
q_{35}^*	q_6	q_{43}
q_{43}^*	q_{35}	q_4

Table 5: Transition Table First NFA with ϵ-closure states and scape state

The regular expression for this automata would be:
$(ab|(abb|b)a((ba)^*(bba)^*)^*)$

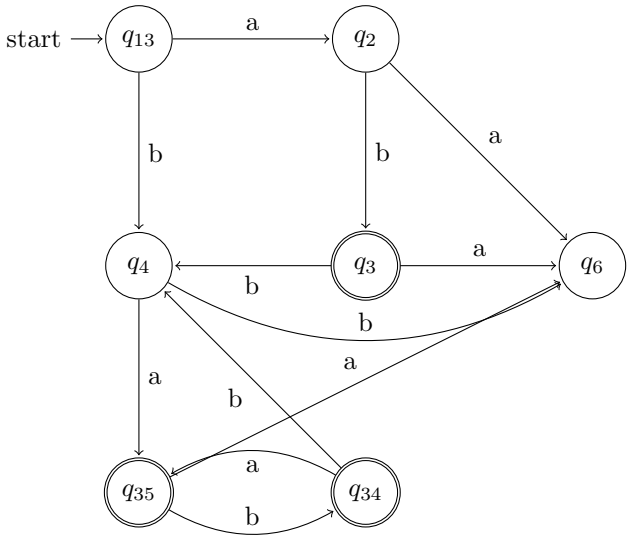

Figure 5: Exercise 2 - Solution 1

Solution Two

In order to find the solution for the second NFA as seen in Figure 6, we will repeat the process followed for the first NFA. We don't want the transitions with ϵ so we can create a table with all the ϵ-closure states. When creating this table, as seen in Table7 it seen that the graph is connected in q_{1234} and q_{234} while the other states are not reachable. In other hand, the ϵ-closure states for 1 and 3 reach all our original nodes, so they can be used to create a NFA.
First we will create a transition table that includes epsilon transitions and then another one with its ϵ-closure states. Using all this information we also get an easy regular expression:None or all b's or at least one a is accepted. $((b)^*|(b)^*a(a|b)^*)$

	A	B	ϵ
q_1^*	q_2, q_4	q_1	q_3
q_2	q_3	q_2	$\emptyset$
q_3^*	q_2	q_3, q_4	q_4
q_4	q_4	$\emptyset$	q_2

Table 6: Transition Table Exercise Two - Second NFA

	A	B
$E(1)^* = q_{1234}$	q_{234}	q_{1234}
E(2) q_2	q_3	q_2
$E(3)^* = q_{234}$	q_{234}	q_{234}
$E(4)^* = q_{42}$	q_{43}	q_2
q_{43}^*	q_{42}	q_{43}^*

Table 7: Transition Table Second NFA with ϵ-closure states

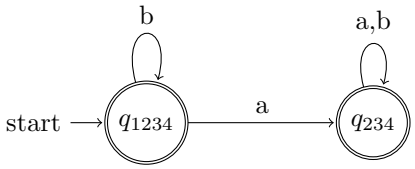

Figure 6: Second DFA solution

Problem 3

Prove that the following languages are or are not regular.

1. L is the language with alphabet {x,y} where the number of x's is equal to the number of y's.
2. L is the language with alphabet {(,)} and balanced parantheses. i.e. (()) is accepted but ()) is not.
3. $L = \{a^2b^3c^4b^1c^m a^n c^4 b^3 a^2 | l, m, n \geq 0\}$

In order to make this assignment shorter I will explain how we can solve this exercise. We can know if a language is not regular if it does not follows the pumping lemma for all the decomposition possibles. If you assume that a language is regular, that means that you have a length "p" can divide a string into xyz and show that xy^iz for $i \geq 0$ is not included in your language, that it does not follows $|y| > 0$, $|xy| \leq p$ or its form does not follow the language. If you find a way to pump the word, then it may be regular.

Part One
The language does not make any reference about the order of x,y and the next exercise is already proving why balanced languages are not regular, so we assume that strings as $x^n y^n$, $y^n x^n$, $(xy)^n$ and $(yx)^n$ are included in the language.
Let's assume the language is regular, the string we are going to prove is $(xy)^n$.

$$
\begin{aligned}
&(xy)^p \\
&\quad p = 5 \\
&\quad w = xyxyxyxyxy \\
&\quad x = xy \\
&\quad y = xy \\
&\quad z = xyxyxy \\
&\quad i = 2 \rightarrow xy^iz = xyxyxyxyxyxyxy \in L
\end{aligned}
$$

We have found a way to pass the pumping lemma, so it cannot prove that the language is not regular. Anyways, for $(xy)^n$ we have a Regular Expression as $(xy)*$ that it can be fixed in the opposite way so the language would be regular for $w \in (x((xy)^*|(yx)^*)^*y)^*|(y((xy)^*|(yx)^*)^*x)^*$. While the x and y are working in pairs, they would be balanced by them selfs, and they can repeat in loops and still be included in the

language.
If the x and y are not working in pairs, like it could be in $x^n y^n$ or $y^n x^n$, then the language would not be regular. The proof to this exercise would be the same as the next part, where a parenthesis would be x and the closing parenthesis y.

Part Two

In this case, the order of the characters is important, they are balanced. So we accept (()) but not))((. I am going to use the pumping lemma in different cases. First case, when the y includes "(" parenthesis.

$$\begin{aligned} i &= 5 \\ xyz &= \{(^6)^6\} \\ x &= \{((\} \\ y &= \{((\} \rightarrow y^5 = \{((((((((((\} \\ z &= \{(())))))\} \\ xy^ic &= \{((((((((((((((()))))))))\} = \{(^{14})^6\} \notin L \end{aligned}$$

The lemma shows that the language is not regular at this point. Let's try when the y part includes "()" parenthesis.

$$\begin{aligned} xyz &= \{(^6)^6\} \\ x &= \{(((((\} \\ y &= \{()\} \rightarrow y^5 = \{()()()()()\} \\ z &= \{)))))\} \\ i &= 5 \\ xy^ic &= \{((((()()()()()))))\} \notin L \end{aligned}$$

In this case the lemma shows a language not regular. I could try the last part with y including only ")", but it will give the same solution that the first part. All the possible decompositions of balanced parenthesis are shown to be not regular.

Part Three

As it has been done until now, let's assume the language is regular, divide it in xyz and apply xy^iz for $i \geq 0$.

$$a^2b^3c^4b^lc^ma^nc^4b^3a^2$$

$$\begin{aligned} i &= 5 \\ a &= \{a^2b^3c^4b^l\} \\ b &= \{c^m\} \\ i = 5 &\rightarrow b = (c^m)^5 \\ c &= \{a^nc^4b^3a^2\} \\ ab^ic &= \{a^2b^3c^4b^lc^{5m}a^nc^4b^3a^2\} \in L \end{aligned}$$

The pumping lemma is not able to show us that this language is not regular. Following the definition of a regular language, we can try to find a regular expression to prove that it is. In our case, the regular expression to prove that the language is regular would be $aabbbcccc(b^*|c^*|a^*)ccccbbbaa$. If we have a regular expression, that means that we can build an automaton that works with our language.

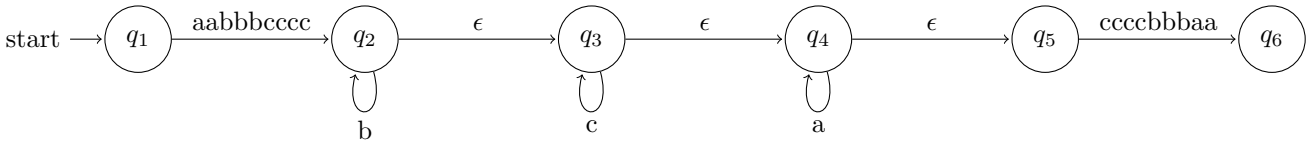

Figure 7: Informal Automaton from the Regular Language

Problem 4

Artificial Intelligence in video games is often implemented using Finite State Machines. Assume that in a turn-based video game there are two types of enemies, a zombie and an archer. At the beginning of each turn an enemy must choose what their action will be. The state is defined by the action the enemy was taking in the previous turn. The initial state for all enemies is idle. A zombie behaves in the following way:

1. If the player is in sight and is alive but not close enough to bite it will chase the player.
2. If the player is in sight, alive and close enough for the zombie to bite, it will do so
3. If the player is not in sight it will simply stay idle.

The process repeats until either the player or the zombie is dead.
The archer behaves a bit differently:

- If a player is in sight, alive and in range but not too close he/she will try to shoot the player.
- If a player is in sight, alive and not in range he/she will try to approach.
- If a player is too close he/she will try to evade
- If his/her hit points are low he/she will try to flee.
- If a player is not it sight he/she will stay idle as well.

This process again repeats until either the player or the archer is dead.
Design a NFA for each of the enemies.

The Zombie
In this first case I would create a NFA, as shown in Figure 8, that would follow a path from the initial state or idle state q_i, to states, "bite" q_b and "chase" q_c until at some point, the player or the zombie dies q_d. After each accepting state from our zombie, the player will take their[1] turn. After the player turn, the zombie will keep moving through the automata until they or the player die. If the player dies, I will assume that a new instance of the game would be created.
Our transitions would be when the game recognizes one of the next:

- A - Player or zombie Alive
- D - Player or Zombie dead
- I - Player in sight
- O - Player out of sight
- C - Player close
- F - Player far

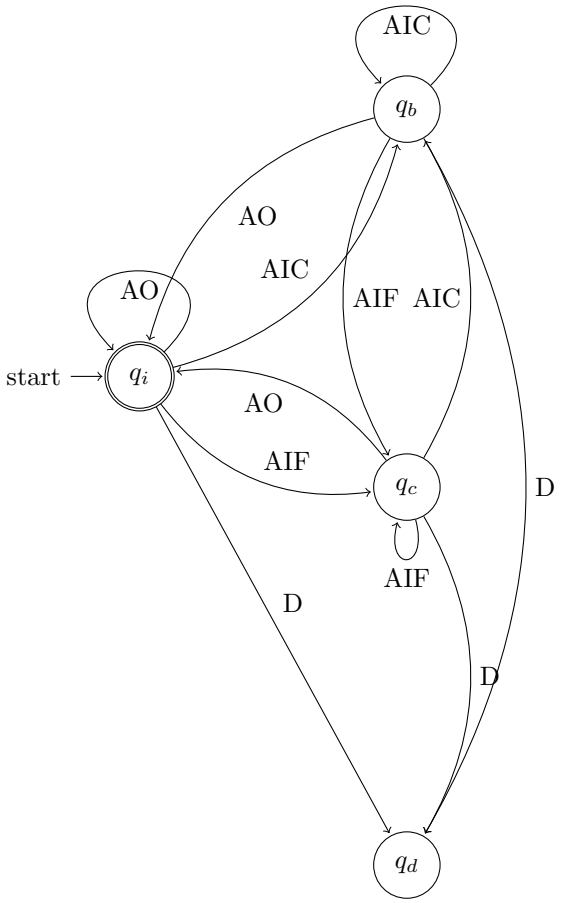

Figure 8: Zombie solution

And they would be applied to the automata showed in the Figure 8.

The Archer

The assumptions I made for the archer are that the archer would flee when low health, but only if the player is visible and alive, otherwise I do not see a reason to flee (if the player is dead or not visible, the archer has no reason to be scared). The accepting states would be, as in the zombie automata, "idle" q_i, "flee" q_f, "approach" q_a, "shoot" q_s, "evade" q_e and "dead" q_d. The transitions would be.

- A - Player or archer Alive
- D - Player or archer dead
- I - Player in sight
- O - Player out of sight
- C - Player close
- R - Player in range
- F - Player far
- HP+ - Archer hit point high

[1]This is not a mistake. They/their are accepted neutral pronouns, also know as "Singular They".

- HP- - Archer hit points low

The automata for the archer is shown in Figure 9.

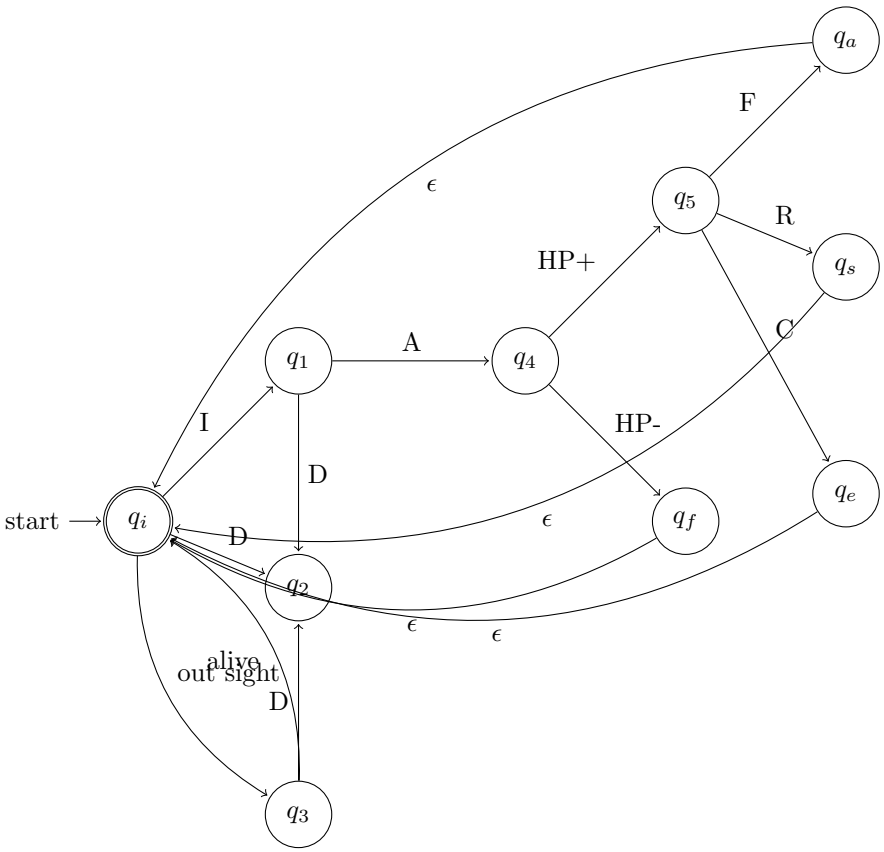

Figure 9: Archer solution

The Archer

Problem 5

An example of an HTML link is the following:

```
<a href="http://example.com/"> Example.com</a>
```

In that particular example the URL is: http://example.com/, and the protocol is http. A URL can be:

- A link to an element with a specified id:

```
<a href="#id">Link to id</a>
```

- Some other protocol, when the href part starts with an alphanumeric string followed by ":" i.e. https:// , ftp:// , mailto: , file: , etc.
- A script: i.e. href="javascript:alert(Hello);"

Problem 5

Design regular expressions that:

- Parse an HTML line and identify all HTML links it contains
- For each URL
  - identify the protocol used if any
  - whether each URL points to an id within the page
  - is a script

Part One
In one way, an html document is composed of strings. The automata should read the string and if it reads the beginning tag for links $<a\ href="$ followed for $(\#|https://|ftp://|mailto:|file:|javascript:)$ any character after and ends up with $>$. I followed the regular expression below for the program in Java. I only used tags, scripts, https and mailto as protocols because I understand that it conveys the idea of the solution for any other kind of link.

```
<\\s*a\\s+.*?href\\s*=\\s*\"(\\S*?)(#|https|script|mailto)(\\S*?)\".*?>
```

Part Three
The language html is not a regular language because the language focus on balanced markup, that means that we do not have memory to make a regular expression when there is a need of balancing nested tags, as proved in exercise 3.2. It is possible to scrap certain conditions from specific tags or patterns. For example, in the exercise it is possible to look for a special pattern like "href" followed ".*?" or even create regular expressions that look for special patterns, but the general language of html in its whole is a context-free grammar, but not regular.
Part Two
The code would be annexed to the Assignment but in case of need, it is showed here.

```
public class Main {
public static void main(String[] args) throws IOException{
BufferedReader bufferedReader = new BufferedReader(new FileReader("src/TEST.html"));
        String line;

        Pattern regexLink = Pattern.compile("<\\s*a\\s+.*?href\\s*=\\s*\"(\\S*?)\".*?>");
        Pattern regexTag = Pattern.compile("(\\S*?)#(\\S*?)");
        Pattern regexHTTPS = Pattern.compile("(\\S*?)https(\\S*?)",Pattern.CASE_INSENSITIVE | Pattern.D
        Pattern regexScript = Pattern.compile("(\\S*?)script(\\S*?)",Pattern.CASE_INSENSITIVE | Pattern
        Pattern regexMailto = Pattern.compile("(\\S*?)mailto(\\S*?)",Pattern.CASE_INSENSITIVE | Pattern


        while ((line = bufferedReader.readLine()) != null) {
         Matcher matchLink = regexLink.matcher(line);
                Matcher matchTag = regexTag.matcher(line);
                Matcher matchHTTPS = regexHTTPS.matcher(line);
                Matcher matchJS = regexScript.matcher(line);
                Matcher matchMailto = regexMailto.matcher(line);

         if(matchLink.find())
         {

         if(matchTag.find()) {
         System.out.println("Tag found: " + line);
                    }
         else if (matchHTTPS.find()) {
         System.out.println("Protocol found: " + line);
         }
         else if (matchJS.find()) {
         System.out.println("Script found: " + line);
         }
         else if (matchMailto.find()) {
         System.out.println("Mailto found: " + line);
         }
         else {
         System.out.println("This link does not follow any protocol: " + line);
         }
        }
        }
        bufferedReader.close();
    }
}
```

Funny Graph

The funny graph works in this way:

1. Move following the black arrows as much times as the character of your string.
2. if that is your last character, the state you are in is the reminder.

3. if it is not your last character and the state has a dashed arrow, move to the state pointed by it.

4. repeat steps until you reach the end

So for example, 365 would move from q_0 to q_3 three times, then back to q_0, then it would move 6 states to q_0, then five positions to q_5, so the remainder of 365 is 5.

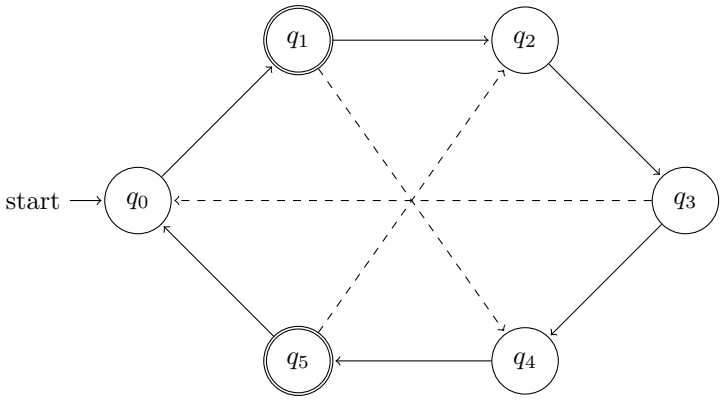

Figure 10: Funny graph of numbers divisible by 6