1. The point here is to a construct a DFA M which accepts L. In order to accomplish this we need a way to keep track of even or odd states, and a way to ensure the game is legal by summing each input such that the only way to reach the accept state is on an even numbered transition where the sum of each n equals k

   The way we will accomplish this is by defining n to be an integer representing the nth turn of the game.

   We will define the number of transitions from each state as l, where $0 < l \leq$ k: where k is the total number of stones

   so then the set of accept states will be F and will be composed of each state where n = an even number and i = k. i will represent the sum of all (l)'s where l is the number of stones chosen on a given turn.

   The notation for each state is as follows $q_{n,l,i}$

   for example $q_{2,\ 7,\ 8}$ would represent the state where player B chooses to take 7 stones on the second turn of the game, while player A took 1 stone on the first turn of the game.

   we formalize the M as the following

   M = $(Q, \Sigma, \delta, q_{n=0,\ l=0,\ i=0}, F)$

   where Q = the set of states: $\{q_{n,\ l,\ i}\}$ such that $n \in thesetofpositiveintegers,\ \ 0 < l \leq k,\ \ \ i = \Sigma$ (from t = 0 to t equals n) of l. That is: for each of the previous states visited starting with state $n_{0,\ 0,\ 0}$ and going to the current state where t = n = n we want the summation of the l, or the total number of stones chosen. In other words the sum of all l in the set of states visited which is equal to $\{q_{0,\ 0,\ 0}, q_{1,\ l,\ i}, ..., q_{n,\ l,\ i}\}$

   Σ is the alphabet {1,2, ... l}

   state $q_{n=0,\ l=0,\ i=0}$ is the start state

   and F is the set of accept states: $\{q_{n,\ l,\ i}\}$such that n is an even number, and i = k

   we define δ (q, l) where q $\in Q$ and l as the number of stones chosen on that turn

   δ $(q_{n,\ l,\ i}, l)$ –¿ $q_{n+1,\ l,\ i+l}$

   ---

2. we can prove that a language A_k exists by simply applying the definition of a DFA and Language.

   For the sake of contradiction, we begin by supposing there is a language A_k which a DFA with k states cannot recognize. We know that a DFA has exactly one possible transition for every element in the language it recognizes. if the language it recognizes has k elements, then k states is enough to suffice to recognize any language in the same class as the one for which every state is utilized to determine if a string is accepted. If every language in the same class is recognized than A_k is recognized. and we have a contradiction demonstrating that there cannot be a DFA with k states which can recognize language A_k. Therefore every DFA with k states can recognize a similar language

To prove that there exists a language A_k such that a machine with k-1 states cannot recognize we will demonstrate a counter example.

Say $A_k = \{0, 1\}$ and a DFA with only 1 state. While the DFA with 2 states can handle the language. A DFA with only one state would not be capable of recognizing this language. for example the single state DFA could not transition to a separate accept state. if the empty string is not accepted the machine would fail to be suitable for recognizing the language.

Similarly, for every k greater than 1, if a DFA must use all k of its states (e.g. a transition is required through each and every one of them), then a DFA with k-1 states would not be able to recognize the same language.

---

3. a)

we begin by creating a function which maps the regular expression as one relation from a newly instantiated start state, s; to a single accept state, a.

$\delta(s, a^*(a \cup b)^*(a^* \cup ab)$

we then create a new state for each substring which comprises a transition.

our viable substrings are $a^* --- (a \cup b)^* --- (a^* \cup ab)$

We replace s with the first of our substrings

Here it is important to note that an empty string will be accepted. Since this is the case our start state will also be an accept state.

if a substring ends with the kleen Star the next substring's state does not need to be a different state. Instead we define the next substring's relation to map from the current substring's state and the current substring's state's relation maps to itself.

since the first two of our three substrings can be empty or contain any number of their elements we have two self-looping relations

$\delta_1(1, a^*) -- > 1$
$\delta_1(1, (a \cup b)^*) -- > 1$

the third substring will transition to the final state.

$\delta_1(1, (a^* \cup ab)^*) -- > a$

we can now replace the union operator with an "or" operator

and since there are two strings which map from state 1 to state 1, we can combine them with an "or" operator as well

this gives us

$\delta_1(1, ((a \vee b)^* \vee a^*) -- > 1$
$\delta_1(1, (a^* \vee ab)^*) -- > a$

we can now see the redundancy in the first relation

if any number of (a or b)'s self loop then we can just draw say for all (a,b) if, in state 1, stay in state one, we also, see that any number of (a)'s will transition to the final state.

$\delta_1(1, (a, b)) - - > 1$
$\delta_1(1, (a^* \vee ab)^*) - - > a$

our final definition of δ is breaking apart the substring ab into two substrings and instantiating a new state to accommodate the new transition

so we will have the string 'a' transition to a new state, 2 from which a b will map to the accept state as well as back to state 1

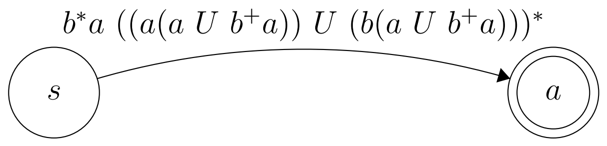
this

$\delta_1(1, (a^* \vee ab)^*) - - > a$

breaks down into

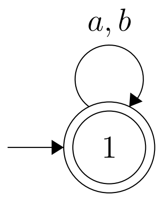
$\delta_1(1, a) - - > a$
$\delta_1(1, a) - - > 2$
$\delta_1(2, b) - - > a$
$\delta_1(2, b) - - > 1$

the following figure represents the NFA N we will simplify and formalize below.



We will notice that the legal string which is accepted by states from 1 to 2 to a will also be accepted by the transition from 1 to 1 to 1. Also the accepted word from 1 to a, will also be accepted by 1 to 1. Therefore, states 2 and a are redundant and can be eliminated.

our final NFA is shown here



we define NFA N

N $= (Q, \Sigma, \delta, 1, F)$

where Q = the set of states: 1

Σ is the alphabet a,b

state 1 is the start state

and F is the set of accept states: 1

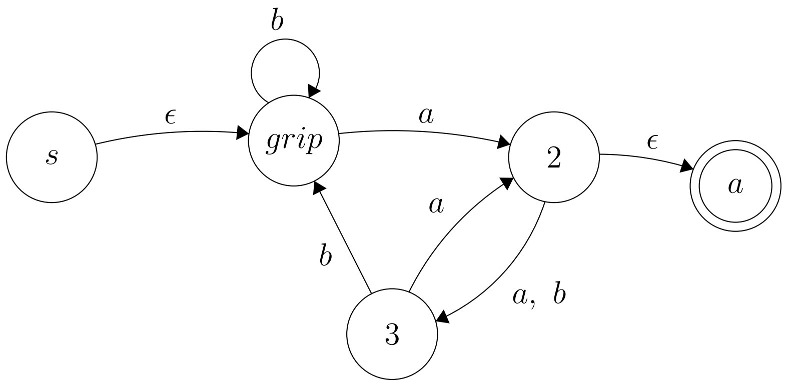
we define $\delta(1, i)$ where i is any element of Σ

$\delta(1,(a,b)) --> 1$

---

b)

step one: add a start state and an accept state. (We have denoted the start state as s, and have omitted the indicator arrow found in diagrams of DFA's and NFA's)



step 2: change state 1 to grip and reconfigure any route going through it such that the state can be ripped out.

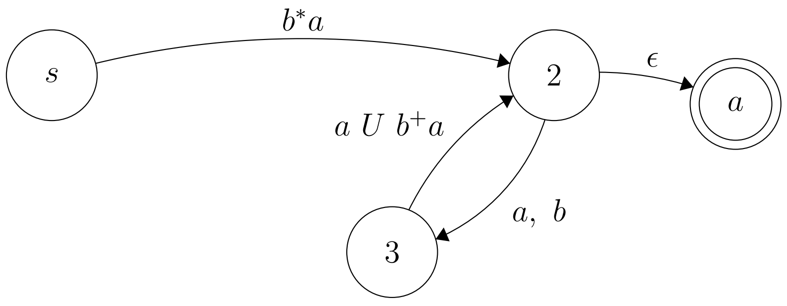
in this case we have

$\delta(3,b) --> 1$
$\delta(1,b) --> 1$
$\delta(1,a) --> 2$

our new relations combine these such that

$\delta(s,b*a) --> 1$
$\delta(3,bb^*a) --> 2$

and our new figure represents these changes



Step 3: we make state 3 grip and reconfigure every route which passes through it.

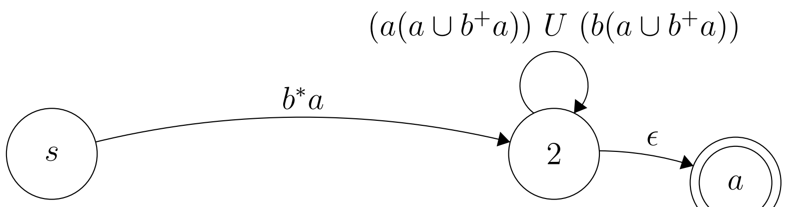
we have

$\delta(3, bb^*a) --> 2$
$\delta(2, a, b) --> 3$

our new relations combine these such that there is a relation from state 2 to itself

$\delta(2, ((a(a \cup b^+a)) \cup (b(a \cup b^+a))) --> 2$

and our new figure represents these changes



Step 4: we make state 2 grip and reconfigure every route which passes through it.

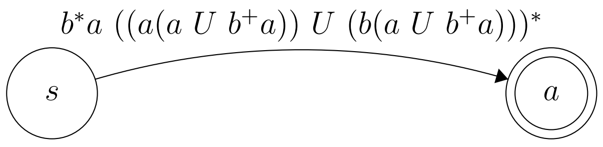
we have

$\delta(s, b^*a) --> 2$
$\delta(2, ((a(a \cup b^+a)) \cup (b(a \cup b^+a))) --> 2$

our new relations combine these such that there is one regular expression transition which comprises every string this GNFA accepts

$\delta(s, (b^*a((a(a \cup b^+a)) \cup (b(a \cup b^+a)))^*)) --> a$

and our final figure represents the GNFA equivalent of the DFA in Figure 1 of homework 3, with exactly 2 states.



---

4. I am not convinced this is a valid definition as it seems circular to me. However, for the sake of finishing my homework I will assume there is some sort of induction which can be applied to the complement operator which will allow for some base case which is simpler than the original Regular Complement Expression R_1 of which we take the complement.

   We must also assume that R exists in an alphabet. We know that alphabets must have at least one element. If it existed in the empty set. Then there would not be a DFA which could accept R and not it's complement or accept the complement of R if it did not accept R, since there is no way to define the set of a complement of the null set from a null set.

to prove that Regular-Complement Expression is equivalent to Regular Expression we must show that for every attribute of the Regular-Complement Expression Definition the same is true for every REgular Expression.

Beings how the first 6 rules are identical to those of the definition of REgular Expression, we can see that they can be applied to both.

The one we must show is that (R_1NOT is a Regular Complement Expression where R_1 is a known to be a regular Complement Expression.

Since we know that there is a DFA which can recognize every regular expression, and by this definition if we can show that every REgular-Complement Expression can be recognized by a DFA, then, a Regular-complement Expression is in the same class as Regular expressions.

for the sake of contradiction we will assume that there is a Regular-Complement Expression which is not recognized by any DFA

but we know that if the definition of R is the same as that of REgular expressions with the addition of being included so long as its complement is included. Since we know it exists in an alphabet, and alphabets consist of a countable number of elements, we know that we can construct a DFA which will accept any language of that Alphabet. As an alphabet must possess at least one element, there will always be a way to construct a DFA which accepts the complement of R. It is as simple as changing every accept-state to a not-accept-state and every not-accept-state to an accept-state.

Therefore we have a contradiction and there are no Regular Complement expressions which are not recognized by any DFA.

Since we have shown every Regular-Complement Expression is recognizable by some DFA, we have proven that regular complement expressions and Regular Expressions are equivalent.

——————————————————-