Chapter 1

Finite State Machines

Finite State Machines? More like infinite town devices

Cringe

1.1 Introduction

So far we have talked about the language features that languages may have. However, now we want to start talking about how we can take features from one language and implement them in another. A naive approach may be something like making a library or some wrapper functions. For a simple example, maybe I wanted to add `booleans` to C. I can just write a `#define` macro for `1` and `0` which we name as `true` and `false`. For a more complicated example if I wanted to add pattern matching in C, then maybe I create a `struct` called `data` which can hold any value which can be pattern matched and a function: `void* match(data* value, int (**patterns)( data* ), void* (**exprs)(data*))` which takes in a piece of data to match, a series of functions that return true if the value matches it, and a series of functions that return some value[1]. This way is terrible and so the typical way to add something is by changing the compiler (Or if we want to go one step further, let's design our own language, which means we need to make a new compiler-HAH!).

1.1.1 Compilers

While this is not CMSC430: Compilers, we need to setup the basis of compilation. We will talk about this more in depth in a future chapter, but here's a quick overview. A compiler is a language translator (typically some higher level programming language to assembly). To translate one language to another, we need to do the following:

- break down the language to bits that hold information
- take those bits and figure out how to store that information in a meaningful way
- take the stored information and map it to the target language
- generate the target language.

The best way to break down the language is to use regular expressions. However, what if your language doesn't have regular expressions? Simple: let's implement regular expressions in a way that we don't need to compile.

1.1.2 Background - Automata Theory

Imagine that we want to create a machine that can solve problems for us. Our machine should take in a starting value or values, a series of steps, and then give us output. Depending on when and who you took CMSC250 with, you already did this.

[1] See Appendix A for a rough implementation //TODO

A circuit or logic gate is the most basic form of this. If our input is values of true and false (1 and 0), let us put those inputs into a machine that *ands*, *ors* and *negates* to get an output value.

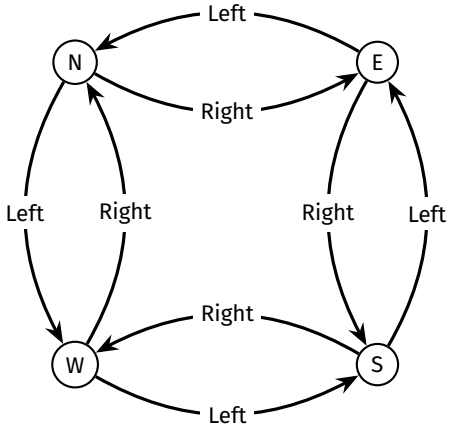
The issue here is we do not have any memory. We cannot refer to things we previously computed, but only refer to things we are inputting in each gate. Once we add a finite amount of memory, we can accomplish a whole lot more and we get what we call finite automata (FA). I use finite automata interchangeably with finite state machine (FSM). Typically, FA is used in the context of abstract theory and FSM is used with the context of an actual machine, but they both refer to the same thing.

Once we start adding something like a stack (infinite memory), we get a new type of machine called push-down automata (PDA) where we theoretically have infinite memory, but we can only access the top of the stack. Lifting this top-only read restriction, we get what is known as a Turing machine[2]. As formalized in the Church-Turing thesis, any solvable problem can be converted in a Turing Machine. A Turing machine that creates or simulates other Turing machines is called an Universal Turing Machine (UTM). Fun fact: Our machines we call computers are UTMs).

All of this is to say that a compiler wants to output a language that is Turing complete, one which can be represented by a Turing machine. Regular expressions on the other hand, describe what we call regular languages, and regular languages can be represented by finite automata. So we will start with FSMs, but know that when we get to compilation, we will need something more.

1.1.3 Finite State Machines

Let's start by modeling a universe and breaking it down to a series of discrete states and actions. Let us suppose that my universe is very small. There is just me, a room and a compass. Suppose I am standing facing north in this room. Let's call this state N. When facing north, I have two options: turn right 90° and face East, or turn left 90° and face West. Let's give these states some names: states E and W respectively. From each of these new positions (facing west or facing east), I could turn left or right again and either end up facing back north or facing south. Let's give the state of facing south a name: S. If I create a graph that represents all possible states and actions of the universe, I could create a graph that looks like:



This graph represents a finite state machine. A physical machine can be made to do these things, but for the most part, we will emulate this machine digitally. We typically define a FSM as a 5-tuple:

- A set of possible actions
- A set of possible states
- a starting state
- a set of accepting states
- a set of transitions

The set of transitions is the set of edges, typically defined as 3-tuple (starting state, action, ending state). To be clear: this is a graph. A transition is an edge, and a state is a node. We haven't seen what a starting or accepting state is, but we will see those in the next section.

[2] Initially called an 'a-machine' or atomic machine by Alan Turing.

The important takeaway from the example above is **Based on where I am (which state), and what action occurs (which edge I choose), I can tell you where I will end up**. So, given an input and a series of instructions, I can give you an output (sound familiar?). For example, if I start at state N, and my instructions are to go left, left, right, left, left, right, right, I can traverse my path ($N- > W- > S- > W- > S- > E- > S- > W$) to know where I am and return it (My output is W here).

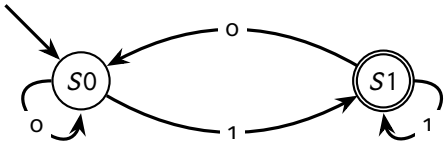
1.2 Regex

So we did this whole thing with graphs and talked about what a machine is and a single-example use case. Let's talk about another use case: regular expressions. For regular expressions, we define a FSM as a 5-tuple very similarly as what we previously had, but instead of actions, we have letters of the alphabet.

So, we have:

- the alphabet (Σ), which is a set of all symbols in the regex
- a set of all possible states (S)
- a starting state (s_0)
- a set of final (or accepting) states (F)
- a set of transitions (δ)

To be clear on types, $s_0 \in S$ and $F \subseteq S$. This is because a FSM can only have 1 starting state (no more, no less), but any number of accepting states (including 0). In the previous example, we can understand the starting state to be N, but we don't really have any accepting states. Let us see an example of a FSM for the regular expression `/(0|1)*1/`.



This machine represents the regular expression `/(0|1)*1/`. Recall that a regular expression describes a set of strings. This set of strings is called a language. Examples of strings in the language described by the regular expression `/(0|1)*1/` would be `"1"`, `"10101"`, and `"0001"`. When we say that a FSM accepts a string, it means that after entering at the starting state (denoted by an arrow with no origin) and traversing the graph after looking at each symbol in the string, we end up in an accepting state (states denoted by a double circle). Let's see an example.

Given the above FSM, suppose we want to check if the string `"10010"` is accepted by the regex. We start out in state $S0$ since it has the arrow pointing to it as the starting state. We then look at the first character of the string: `"1"` and consume it. If we are in state $S0$ and see a `"1"`, we will move to state $S1$. We then look at the next second of the string (since we consumed the first one): `"0"` and consume it. Since we are in state $S1$, if we see a `"0"`, then we move to state $S0$. We then proceed to traverse the graph in this manner until we have consumed the entire string. The traversal should look something like

```
S0 -1-> S1 -0-> S0 -0-> S0 -1-> S1 -0-> S0
```

Since we end up at state $S0$, and $S0$ is not an accepting state (it does not have a double circle), then we say this machine (this regular expression) does not accept the string `"10010"`. Which is true, this regex would reject this string.

On the other hand if traversed the graph with `"00101"`, our traversal would look like

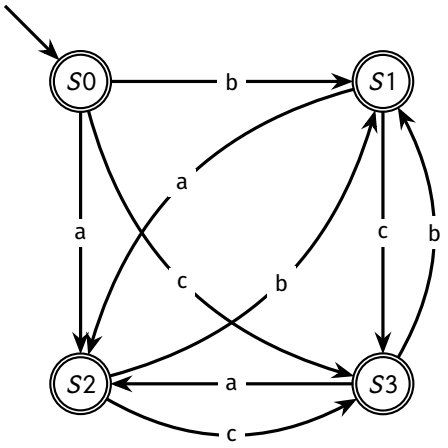
```
S0 -0-> S0 -0-> S0 -1-> S1 -0-> S0 -1-> S1
```

and we would end up in state $S1$ which is an accepting state. So we could say that the machine (the regular expression) does accept the string `"00101"`.

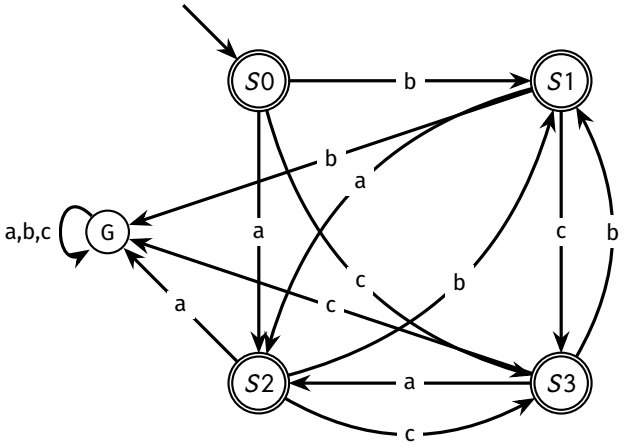
One final thing: a FSM for regex only will tell you if a string is accepted or not[3]. It will not do capture groups, will not tell you if the input is invalid, it will only tell you if the string is accepted (in the case of invalid input, it will tell you the string is not accepted).

1.3 Deterministic Finite Automata

All FSMs can be described as either deterministic or non-deterministic. So far we have seen only deterministic finite automata (DFA). If something is deterministic (typically called a deterministic system), then that means there is no randomness or uncertainty about what is happening (the state of the system is always known) [4]. For example, given the following DFA:



Now this graph is missing a few states (one really). What happens when I am in state $S1$ and I see a `"b"`? There is an implicit state which we call a `"garbage"` or `"trash" state`. A trash state is a non-accepting state where once you enter, you do not leave. There is an implicit one if you are trying to find a transition symbol or action which does not have output here. That is, there is an edge from $S1$ to the `garbage state` on the symbol `"b"`. There are also transitions to the garbage state from $S2$ on `"a"` and $S3$ on `"c"`. If we really wanted to draw the garbage state in, we could like so (but there really is no need to do so):



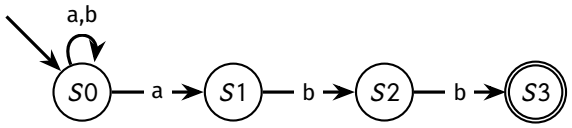
Regardless if we indicate a trash state or not, no matter which state I am in, I know exactly which state I will be in at any given time.

[3] that is, if the string is in the language the regular expression describes. Any language a regular expression can describe is called a Regular Language

[4] Determinism in philosophy is about if there is such a thing on free will and I would definitely recommend reading David Hume's and David Lewis's take on causality

1.4 Nondeterministic Finite Automata

The other type of FSM is a nondeterministic finite automata (NFA). Nondeterministic in math terms means that is something that is some randomness or uncertainty in the system. A NFA is still a FSM, the only difference is what are allowed as edges in the graph. There are 2 of them. Let's talk about one of them now. Consider the following FSM:

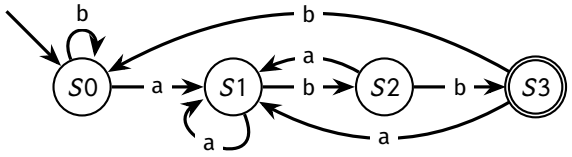


This machine represents the regular expression `/(a|b)*abb/`. There is still a starting state, transitions, ending states, all the things we see for a FSM. However, there is something interesting when we look at the transitions out of $S0$. If I am looking at the string `"abb"`, then when I am traversing, do I go from $S0$ to $S1$ or do I loop back around and stay in $S0$? In fact, there are two ways I could legally traverse this graph:

```
S0 -a-> S1 -b-> S2 -b-> S3
S0 -a-> S0 -b-> S0 -b-> S0
```

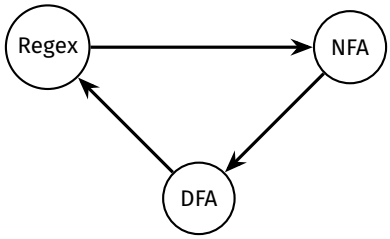
Since the traversal of the graph is uncertain, we call this non-deterministic. To check acceptance for an NFA, we have to try every single valid path and if at least one of them ends in an accepting state, then we accept the string. Since we have to check all possible paths, you can imagine that this is quite a costly operation on an NFA. Additionally, all NFAs have a DFA equivalent. So why use an NFA?

Let us first consider why we are using a FSM. We want to implement regex. To convert from a regular expression to a DFA can be difficult. Consider the above NFA for `/(a|b)*abb/`. Now consider the following DFA for the same regex:



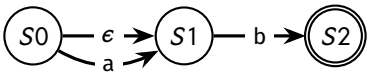
It is much easier to go from a regular expression to an NFA than it is to go from a regular expression to a DFA. Additionally, NFAs, because they cen be more condensed, are typically more spatially efficient than their DFA counterpart. However, there is of course a downside: NFA to regex is difficult, and checking acceptance is very costly. However, NFA to DFA is a one time cost and its less costly to check acceptance on a DFA. Additionally, going from a DFA to a regular expression is much easier. Now keep in mind, technically all DFAs are NFA, but not all NFAs are DFAs.

To visualize this, we typically draw the following triangle



We will talk about how to convert between all of these in a bit, but before we get too far ahead of ourselves, we need to consider the other difference an NFA has over a DFA: epsilon transitions.

An ϵ-transition is a "empty" transition from one state to another. If we think of our graph as one where the edges transitions are the cost to traverse that edge, then an ϵ-transition is an edge that does not cost anything to traverse (it does not consume anything). Consider the following NFA:



If I wished to check acceptance of the string `"b"`, then my traversal may look like:

```
S0 -ε-> S1 -b-> S2
```

Whereas my traversal of the string `"ab"` may look like:

```
S0 -a-> S1 -b-> S2
```

Knowing this, you can see that this machine represents the regular expression: $/a?b/$.

1.5 Regex to NFA

Now that we know what a FSM, NFA and a DFA is, then we can loop back around to our initial goal: implementing regular expressions. In order to do this, we will of course need to build an NFA. To do so, we need to think about the structure of a regular expressions. That is, we need to consider what the grammar of a regular expression. We will talk about grammars in a future chapter, but a grammar is basically rules that dictate what makes a valid expressions. Here is the grammar for a regular expression:

$$\begin{aligned} R \rightarrow \quad & \emptyset \\ & |\epsilon \\ & |\sigma \\ & |R_1 R_2 \\ & |R_1 | R_2 \\ & |R_1^* \end{aligned}$$

All this says is that any Regular expression is either

- something that accepts nothing ($\emptyset$)
- something that accepts an empty string (ϵ)
- something that accepts a single a single character (σ)
- a concatenation of 2 Regular expressions ($R_1 R_2$)
- One regular expression or another regular expression ($R_1 | R_2$)
- A Kleene Closure of a regular expressions (R_1^*)

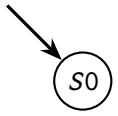
To convert from a regular expression to a NFA, all we need to figure out how to represent each of these things as an NFA. Since this grammar is recursive, we will start with the base cases, and then move on to the recursive definitions.

1.5.1 Base Cases

There are three base cases here: $\emptyset, \epsilon, \sigma$. Let's look at each of these.

The $\emptyset$

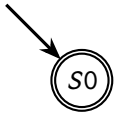
The empty set is a regex that accepts nothing (the set of strings (the language) it accepts is empty). This machine can be constructed as just the following:



Even if Σ has characters in it, we just have a single state, and upon seeing any value, we get a garbage state.

The empty string (ϵ)

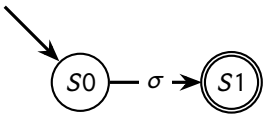
The regular expression that only accepts the empty string means the set of accepted strings is $\{""\}$. This set is not empty so it is different from $\emptyset$. This machine can be constructed as the following:



Even if Σ has characters in it, we just have a single state, and upon seeing any value, we get a garbage state since we are not accepting any strings of with a size greater than 0.

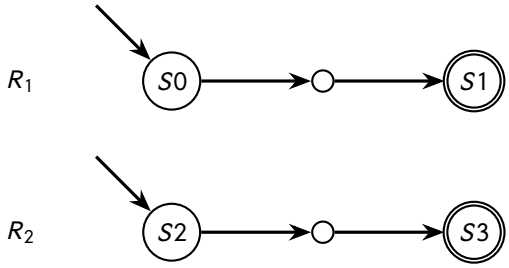
A single character σ

The last base case is a regular expression which accepts only a single character of the alphabet. So if $\Sigma = \{"a", "b", "c"\}$, then we are looking for a regular expression that describes only `/a/`, `/b/`, or `/c/`. We call a single character σ. This machine can be represented in the following manner:



1.5.2 Concatenation ($R_1 R_2$)

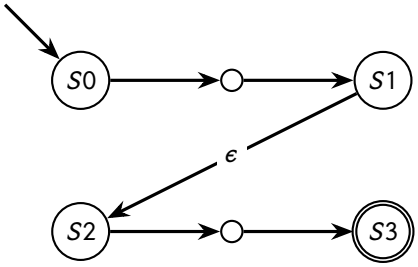
Now that we have our base cases, we can begin to start showing how to do a recursive operation. Aside from the $\emptyset$, each base case has a starting state and an accepting state (sometimes these are the same state). Now since the $\emptyset$ is empty, all the recursive definitions cannot rely on it, so we don't need to really include it as a base case for these recursive calls. Hence, let us assume we have some previous regular expressions R_1 and R_2 that have a starting state and an accepting state.



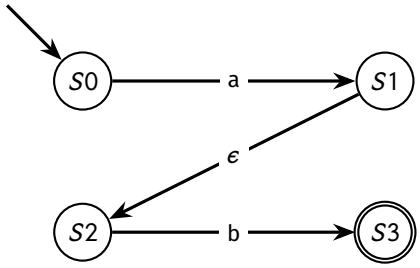
Now we don't know what regular expression R_1 and R_2 are, just that they have 1 starting state, and 1 accepting state. The small, unlabeled nodes here just represent any internal nodes could exist (if any).

To concatenate these two together, it means that if L_1 is the language corresponding to R_1 and L_2 is the language corresponding to R_2, then we are trying to describe L_3 which can be represented as $\{xy | x \in L_1 \wedge y \in L_2\}$. For example, if R_1 is `/a/` and R_2 is `/b/`, then $L_1 = \{"a"\}$ and $L_2 = \{"b"\}$. This means that $L_3 = \{"ab"\}$.

So to take our previous machines, and create a new machine which represents our concatenation operations, we can do so by looking at what our new final states are, and how we get an ordering. Our new machine should have 1 final state which should be the same as our R_2 machine, and should have a way to get from R_1 to R_2 without costing us anything. By implementing these two steps, we get the following machine:



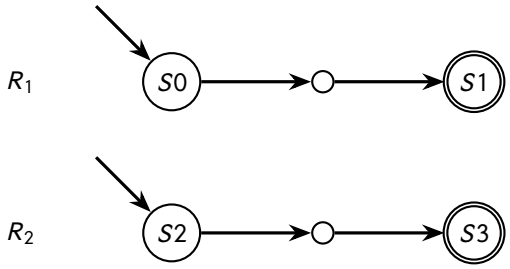
To take our previous example of R_1 being /a/ and R_2 being /b/, when we want to concatenate these machines we get the following machine:



Now, I would say that we are done at this point, as we have a machine that accepts only the concatenated string, with one starting state and one final state (having only one final state is not a restriction of a FSM, but having only one allows us to inductively build our machines here). If we really wanted to, we could optimize the machine a little bit, but it is not necessary.

1.5.3 Branching ($R_1|R_2$)

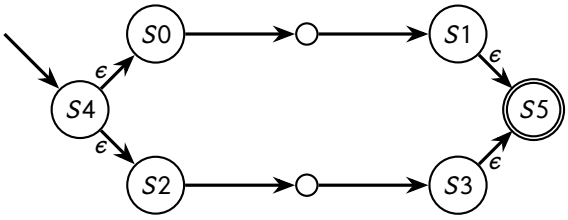
Branching or union is the next recursive definition and requires a bit more work than our concatenation. Again, let us assume that we have some previous regular expressions R_1 and R_2 that have a starting state and an accepting state.



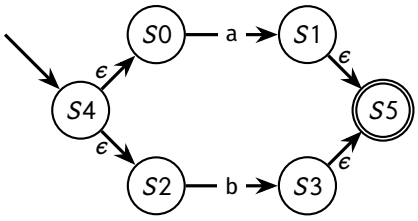
Again we don't know what regular expressions R_1 and R_2 are, just that they have one starting state and one accepting state.

To union two regular expressions together it means that if L_1 is the language corresponding to R_1 and if L_2 is the language corresponding to R_12, then we are trying to describe L_3 which can be represented as $\{x|x \in L_1 \vee x \in L_2\}$. For example if R_1 is /a/ and R_2 is /b/, then $L_1 = \{"a"\}$ and $L_2 = \{"b"\}$. This means that $L_3 = \{"a", "b"\}$.

So to take our previous machines and create a new machine which represents our union operation, we can do so by considering what it means to traverse the graph such that either previous machine is valid. Again, to keep our inductive properties, we need one starting state and one accepting state. Here is where the tricky part comes. We need to make sure that both R_1 and R_2 are accepted with a single accepting state, as well as making sure we can traverse R_1 or R_2 with only 1 start state. The easiest way to do so is by making use of ϵ-transitions with 2 new states. Here is the resulting machine:



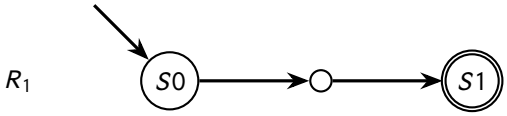
This new machine still has one starting state, and one accepting state which means we can inductively build larger machines, and the ϵ-transitions allow us to chose either path or go to the accepting state without consuming anything. To take our previous example of R_1 being /a/ and R_2 being /b/, when we want to union these machines we get the following machine:



Again, we could choose to optimize this machine, but it's not necessary.

1.5.4 Kleene Closure (R_1^*)

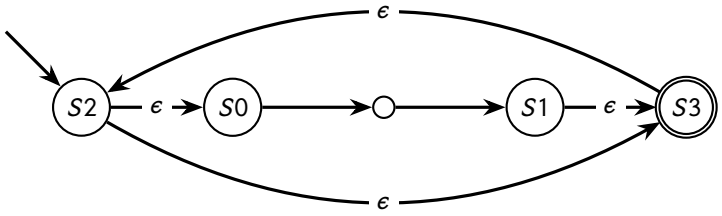
Kleene closure gives us the ability to repeat patterns infinitely many times and is the last recursive definition of a regular expression. Despite having the ability to infinitely repeat, it will look very similar to our union machine. Additionally, this is the only recursive definition that does not rely on two previous regular expressions, so here we only need to assume that we have some previous regular expressions R_1 that has a starting state and an accepting state.



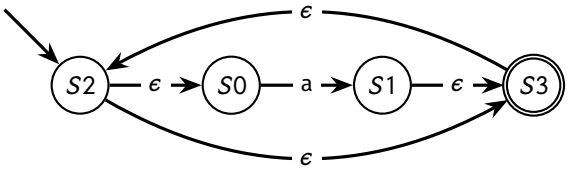
Again we don't know what regular expression R_1 is, just that it has one starting state and one accepting state.

The Kleene Closure of a language, is just analogous to the language's regular expression with the $*$ operator. That is, if L_1 is the language corresponding to R_1 then we are trying to describe L_2 which can be represented as $\{x | x = \epsilon \vee x \in L_1 \vee x \in L_1L_1 \vee x \in L_1L_1L_1 \vee \dots\}$. For example, if R_1 is `/a/` then $L_1 = \{"a"\}$ and we are looking for `/a*/` or $L_2 = \{"", "a", "aa", "aaa", \dots\}$.

So if we take our previous machine, we need to consider how we can accept the empty string as well as any number of repeats of a regular expression. The trick for this is in the definition. We are essentially 'or'ing together the same regular expression repeatedly. So will need to designate a new start state and a new ending state. Doing so will result in the following machine:



Here is where the ϵ-transitions become really important. To accept the empty string, we just use an ϵ-transition to move from $S2$ to $S3$. For repeated values, we can just use the ϵ-transitions from $S3$ to $S2$. Let's look at the previous example of R_1 being `/a/` and seeing the resulting Kleene closure, but also how we would traverse it. The machine would look like:



If I wanted to accept the empty string my traversal would look like

```
S2 -ϵ-> S3
```

If I wanted to accept `"a"`, then my traversal would look like

```
S2 -ϵ-> S0 -a-> S1 -ϵ-> S3
```

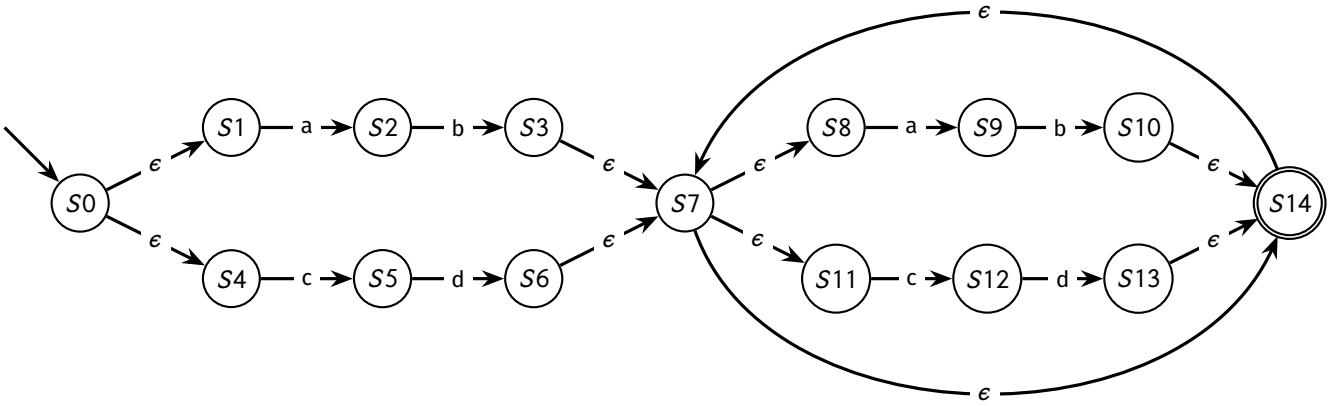
If I wanted to accept `"aa"`, then my traversal would look like

```
S2 -ϵ-> S0 -a-> S1 -ϵ-> S3 -ϵ-> S2 -ϵ-> S0 -a-> S1 -ϵ-> S3
```

We can of course optimize this machine, but again it is not necessary.

1.5.5 Example

For a quick example, if we wanted to make the NFA for the regular expression: `/(ab|cd)+/` the machine (with a few optimizations due to space) would look like:

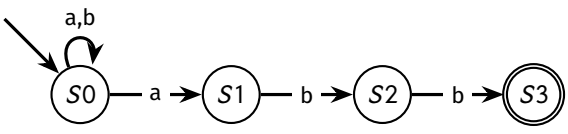


We could optimize this even further, but not really needed. Additionally to see without any optimizations and for a step-by-step, you can see Appendix B (//TODO).

1.6 NFA to DFA

Now that we know how to convert from a regular expression to a NFA, we should talk about to how to convert from a NFA to a DFA. The reason being is that checking for acceptance on an NFA can be really costly and typically you will be calling `accept` multiple times on a machine. So instead of calling `nfa-accept` *n* times, which is a costly operation, you should convert the NFA to a DFA (which is still costly, but done once), so you can then call `dfa-accept` *n* times which is a very cheap operation.

So lets start out by considering the difficulty of `nfa-accept`. Consider the following NFA:

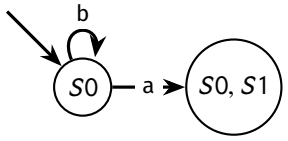


When we want to check acceptance, we need find all possible paths and check if at least one accepts it. That is when checking if the machine accepts `"aabb"`, we need to check all of the following paths:

```
S0 -a-> S1 -a-> Garbage
S0 -a-> S0 -a-> S0 -b-> S0 -b-> S0
S0 -a-> S0 -a-> S1 -b-> S2 -b-> S3
```

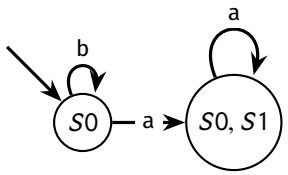
Then since one of them ends in the accepting state, then we can say this machine accepts this string. Notice that we are essentially doing a depth-first-search here which can be terrible if we got an NFA that has a Kleene closure because we get an infinite depth. Additionally, the crux of the problem is that we have no idea which state I am in when I first see the `"a"`. I could either go from `S0 -a-> S0` or I could go from `S0 -a-> S1`. This uncertainty is what makes an NFA non-deterministic. The solution here is to create a new state which represents this uncertainty.

To demonstrate this idea, let's add a state that says "I don't know if I am in state $S0$ or $S1$". Notice this will only happen when we start by looking for an `"a"`. Additionally, when we start with a `"b"`, we know that we have to stay in state $S0$ (that is, if we are in state $S0$ and see a `"b"`, we can only go to $S0$. There is no uncertainty here. But if we are in state $S0$ and see a `"a"`, we could be in $S0$ or $S1$).



Now while we have a new state that shows possible states I could be in after seeing a `"a"`, I then need to figure out what to do next. That is, if I am in this new state $S0$, $S1$, then what happens if I see a `"a"` or a `"b"`?

If this state shows where I could possibly be, then we need to consider both possibilities [5]. So going back to the original NFA, if I am in state $S0$ and see a `"a"`, I could go to state $S0$ or $S1$. Additionally, (looking at the original NFA), if I am in state $S1$ and I see a `"a"`, then I can't go anywhere but the garbage state. So not including the garbage state, we can say that regardless of being in $S0$ or $S1$, if I see an `"a"`, then I have to either be in state $S0$ or $S1$. Well we already have a state that represents this possibility so we can just add the following transition:



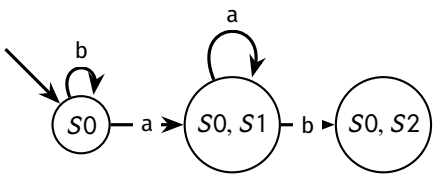
If this is confusing, consider the following logical argument:

$$\begin{array}{ll} & p \Rightarrow q \\ & s \Rightarrow r \\ & p \vee s \\ \hline \therefore & q \vee r \end{array}$$

From CMSC250 we know this is a valid logical argument (known as constructive dilemma). The same applies here. If $S0$ and `"a"` leads to $S0$ and $S1$, and $S1$ and `"a"` leads nowhere (except the garbage state) then we will and up in either $S0$ or $S1$ (or the garbage state).

$$\begin{array}{ll} & S0 \Rightarrow \{S0, S1\} \\ & S1 \Rightarrow \varnothing \\ & S0 \vee S1 \\ \hline \therefore & \{S0, S1\} \cup \varnothing = \{S0, S1\} \end{array}$$

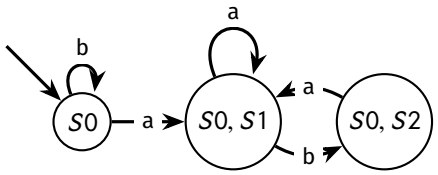
Anyway, that was a slight sidetrack. We next need to consider if we are in $S0$, $S1$ and we see a `"b"`. The above logic applies. If I am in state $S0$ and see a b (looking at the original NFA), then I will end up in state $S0$. If I am in state $S1$ of the original NFA and see a b, then I will end up in state $S2$. It is uncertain which state I will be in though so let's add a new state that represents being in $S0$ or $S2$.



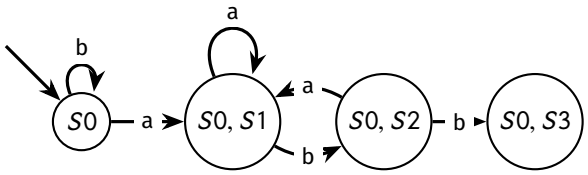
[5] In "laymen's" terms, we are in quantum superposition, that is we are in both $S0$ and $S1$ at the same time

But now the issue continues, if I am in state $S0$, $S2$, and see a symbol, I do not know where I should go. SO let's continue with considering if I was in $S0$ or $S2$ and seeing either a "a" or a "b".

If I am in state $S0$ and see a "a", then I am either in $S0$ or $S1$. If I am in state $S2$ and see a "a", then I can go nowhere (except the garbage state). So if I am in either state $S0$ or $S2$ and see a "a", then I will end up in either $S0$ or $S1$ (or garbage). Let us add this transition.

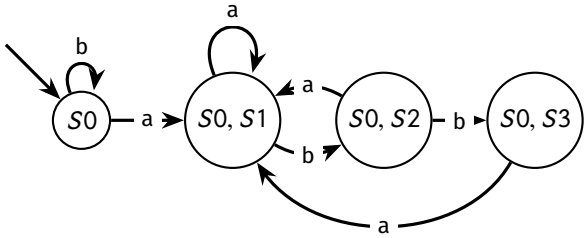


Now if I am in state $S0$ and see a "b" then I can only go to state $S0$. If I am in state $S2$ and see a "b", then I can only go to state $S3$. So if I am in either state $S0$ or $S2$, then I will end up in either $S0$ or $S3$. Here is another place of uncertainly so let us add this new state with transition.

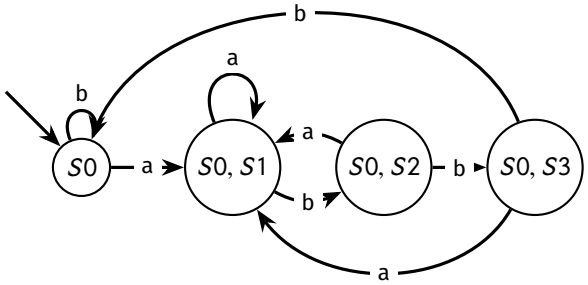


Again, the same issue arises. If I am in state $S0$, $S3$, what happens if I see a "a" or "b"? Well we will have to calculate this like we did with the other states.

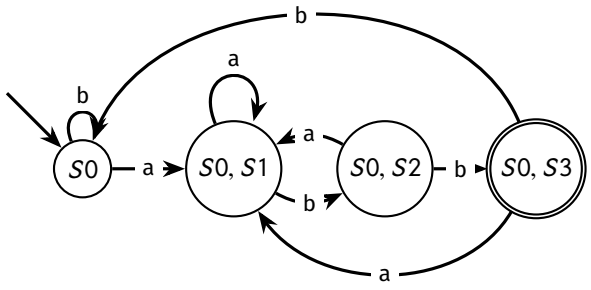
If I am in state $S0$ and see a "a", then I could either be in $S0$ or $S1$. If I am in state $S3$ and see a "a", then I can go nowhere (except a trash state). So if I am in either state $S0$ or $S1$, then I can only end up in $S0$ or $S1$. Let us add this transition.



Now if I am in state $S0$ and see a "b" then I can only go to state $S0$. If I am instate $S3$ and see a "b", then I cannot go anywhere (except garbage). So if I am in state $S0$ or $S3$, if I see a "b", I can only really go to $S0$. So let us add this transition:



Now notice that this time around, we did not add any new states and we know where we want to go from each state. That is, there is no ϵ-transitions, and no state has multiple outgoing edges on the same symbol. By not having these two things, we have created a DFA from our initial NFA. There is just one final step: which states should be our accepting states? If the whole thing is based on possibility being in a state, then it should follow that any state which represents a possible accepting state should in turn, be an accepting state. In our original NFA, $S3$ was the only accepting state, so we look at all states of this DFA and mark any state which represents the possibility of being in $S3$ as an accepting state.



If you also go back a few pages, this machine is identical to the DFA we said corresponded to our NFA. Wild.

1.6.1 NFA To DFA Algorithm

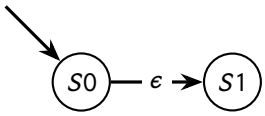
So let us convert what we just did to an algorithm.

To do so, we will need to define 2 subroutines: ϵ-`closure` and `move` as well as define our NFA. Let us use the same FSM defintion we have been using: (Σ, S, s_0, F, δ). Let us give some types as well:

- Σ: `'a list`, a list of symbols
- S: `'b list`, a list of states
- s_0: `'b`, a singe state ($s_0 \in S$)
- F: `'b list`, a list of state we should accept ($F \subseteq S$)
- δ: `('b * 'a * 'b) list`, a list of transitions from one state to another, (`(source, symbol, destination)`).

ϵ-Closure

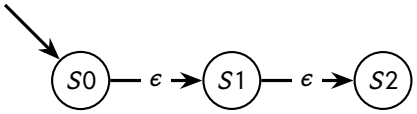
Now to our functions. Recall that we want to figure out which states can be grouped together. ϵ-closure is a function that helps figure this out in terms of ϵ-transitions. THe previous example did not have any ϵ-transitions but consider:



If I am in state $S0$, I could also possibly be in state $S1$ as well. So ϵ-closure if a function that helps us figure out where can we go using only ϵ-transitions. Now the term closure should give us a hint as to what we want to do. We want to figure out what states are closed upon ϵ-transitions. It is important to note that any state can reach itself via an ϵ-transition. So here is the type of ϵ-closure:

- `e-closure: (NFA -> 'b list -> 'b list)`, given a list of states, return a list of states reachable using only ϵ-transitions

To see an example, let us consider the following machine:



If I were to call ϵ`-closure nfa [S0]` I should get back `[S0;S1;S2]`. The best way to do so is by iterating through δ and checking where you can go to anywhere in the input list. Then recursively calling ϵ-closure on the resulting list. That is:

```
e-closure nfa [S0]
// Looking at S0 I can only go to S0 and S1 via an epsilon transition
[S0;S1]
// looking at S0 I can go to S0 and S1, looking at S1 I can go to S1 and S2
[S0;S1;S2]
// looking at S0 I can go to S0 and S1, looking at S1 I can do to S1 and S2,
// looking at S2 I can go to S2
[S0;S1;S2]
// I got no new states, my output matches my input, I am done
```

Here since, my output matches my output then I can return this list and be done. This type of algorithm is called a fixed point algorithm.

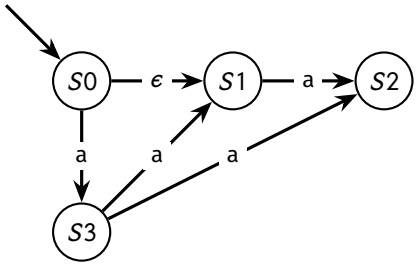
The actual pseudocode is something like:

```
NFA = (alphabet, states,start,finals,transitions)
e-closure(s)
  x = s
  do
    s= x
    x = union(s,{dest|(src in s) and (src,e,dest) in transitions})
  while s!= x
  return x
```

On the other hand, `move` is going to just see where you can do based on a starting state and a symbol. It's type is

- `move: (NFA -> 'a -> 'b -> 'b list)`, given a state and a symbol, return a list of states I could end up in.

It is important to note that you should not perform ϵ-closure at any point during a move. For an example, consider the following machine:



If we were to call `move "a" S0`, then we should get back `[S3]`. Yet if we were to call `move "a" S3` we should get back `[S1;S2]`. This one is pretty straightforward. Just iterate through δ to figure out what your resulting list should be.

NFA to DFA Pseudocode

Now that we have everything defined, need to take the process we had and create an algorithm. Going back to the NFA to DFA example, on each step we had to figure out where we could be upon each symbol in the alphabet. So our pseudocode should look like:

```
NFA = (a, states, start,finals,transitions)
DFA = (a, states, start,finals,transitions)
visited = []
let DFA.start = e-closure(start), add to DFA.states
while visited != DFA.states
  add an unvisited state, s, to visited
  for each char in a
    E = move(s)
    e = e-closure(E)
    if e not in DFA.states
```

```
      add e to DFA.states
    add (s,char,e) to DFA.transitions
DFA.final = {r| r∈ DFA.states and ∃ s ∈ r and s ∈ NFA.final}
```

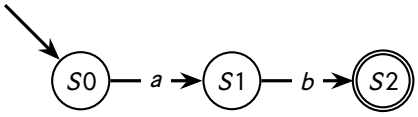
For a full in-depth example, see Appendix C //TODO

1.7 DFA to Regex

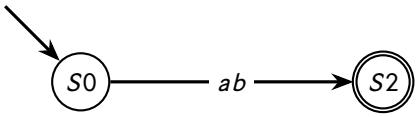
This algorithm to turn any DFA to a regex does not guarantee a nice or readable regex, but it does at least give you a regex. That's all that's important yeah?

We know that every regex is a union, concatenation, or Kleene star of smaller regular expressions. We know that the base regular expression we ultimately build upon is a single character or empty string. If we consider that each transition is a single character, we can combine transitions in a particular way to build a singular regex.

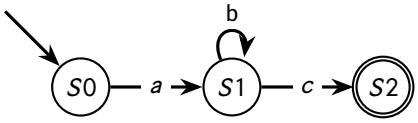
The overarching idea is to take one state out of the machine at a time while replacing the input and output transitions as combinations of each other. Let's see an example:



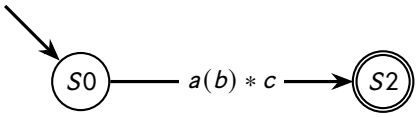
If I took out the $S1$ state, then we can concatenate the input `'a'` with the output `'b'` and end up with a single transition `'ab'` from $S0$ to $S2$.



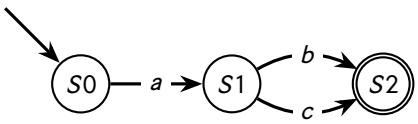
If a state has a self-cycle, then we modify this transition with the Kleene operator, and then insert it between the input and output.



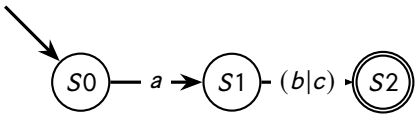
If I took out the $S1$ state, then we can concatenate the input `'a'` with Kleen of the self-cycle `'b'` and concatenate the result with the output `'b'` and end up with a single transition `'a(b*)c'` from $S0$ to $S2$. Consider why this works. We first start with `'a'` and then we could see any number of `'b'`s or see no `'b'`s. Then we end with the `'c'` character. Thus, self-cycles are modified with the Kleene operator.



Lastly, if there are multiple edges from state x to state y, then we will 'OR' these transitions together. This is because there are multiple ways to get from x to y.

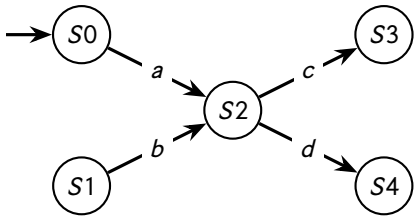


I can combine the `'b'` transition with the `'c'` transition to get the `'b|c'` regular expression. This is because I could take `'b'` or `'c'` transition.

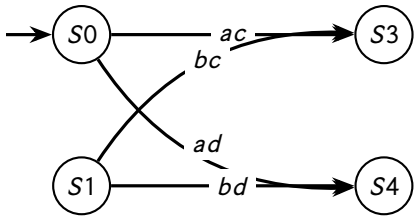


Lastly when we have multiple inputs or multiple outputs (or both) that go to different places, we need to treat each combination of input and output transitions as a new transition we make.

Consider in the following where we have $S2$ as a midpoint for $\overline{S0S3}$, $\overline{S0S4}$, $\overline{S1S3}$ and $\overline{S1S4}$.



We now need to consider the path for each of those connections. For $\overline{S0S3}$, we have `'ac'`. For $\overline{S0S4}$, we have `'ad'`. For $\overline{S1S3}$, we have `'bc'`. For $\overline{S1S4}$, we have `'bd'`. Thus we need to make these 4 connections.



Now to actually run this algorithm, you will need to wrap your DFA in a new final state and a singular final state. From there, you then remove every internal state (states that are not the start nor the final). When you have only 1 transition left (from the new start to the new singular final), then you should have your regular expression.