# CS-440 : Notes

Charlie Stuart : src322

Nowak : Spring 2021

**IN PROGRESS!!!!**

Note: Sections are numbered with my heart

## Contents

## 1 Other Resources

**Turing Machines :** A Turing Machine made with a white board `https://youtu.be/E3keLeMwfHY`

# 2 Math Review

## 2.1 Set Theory

From pages 3-7 in *Introduction to the Theory of Computation* by Michael Sipser

**Set :** A group of objects represented as a unit
**Element :** An object in a set
**Member :** An object in a set
**Multi Set :** An set containing an element that occurs multiple times
**Subset :** A set that consists of elements that exist in a different set
**Proper Subset :** A set that is a subset of another set, but not equal
**Infinite Set :** A set of infinitely many elements
**Empty Set :** A set of no elements
**Singleton Set :** A set of one elements
**Unordered Pair :** A set of two elements
**Sequence :** A set in a specific order
**Tuple :** A finite set
**k-Tuple :** A tuple of $k$ elements
**Ordered Pair :** A 2-tuple
**Power Set :** All the subsets of A
$\in$ : Is a member of
$\notin$ : Is not a member of
$\subset$ : Is a proper subset of
$\not\subset$ : Is not a proper subset of
$\subseteq$ : Is a subset of
$\nsubseteq$ : Is not a subset of
$\cup$ : Union of two sets
$\cap$ : Intersection of two sets
$\times$ : Cross product of two sets
$\mathbb{N}$ : Set of natural numbers
$\mathbb{Z}$ : Set of integers
$\mathbb{Q}$ : Set of rational numbers
$\mathbb{A}$ : Set of algebraic numbers
$\mathbb{R}$ : Set of real numbers

A set is defined in a few ways

| | |
|---|---|
| $S = \{7, 21, 57\}$ | Finite Set |
| $S = \{1, 2, 3...\}$ | Infinite Set of all natural numbers $\mathbb{N}$ |
| $S = \{7, 7, 21, 57\}$ | Multi Set |
| $S = \emptyset$ | Empty Set |
| $S = \{5\}$ | Singleton Set |
| $S = \{5, 3\}$ | Unordered pair |
| $S = \{n \mid n = m^2 \text{ for some } m \in \mathbb{N}\}$ | Set of perfect squares |

The union of two sets is the same as an OR operator in boolean algebra. It's all the elements in both sets.

$$\begin{aligned} A &= \{1,2,3\} \\ B &= \{3,4,5\} \\ A \cup B &= \{1,2,3,4,5\} \end{aligned}$$

The intersection of two sets is the same as an AND operator in boolean algebra. It's all the elements that appear only in both sets.

$$\begin{aligned} A &= \{1,2,3\} \\ B &= \{3,4,5\} \\ A \cap B &= \{3\} \end{aligned}$$

The Cartesian product, or cross product, of two sets is the set of all ordered pairs where the first element is a member of the first set and the second element is a member of the second set for every combination.

$$\begin{aligned} A &= \{1,2\} \\ B &= \{x,y,z\} \\ A \times B &= \{(1,x),(2,x),(1,y),(2,y),(1,z),(2,z)\} \end{aligned}$$

A relation is a subset of $X \times X$. It lists all pairs that are related by $R$

$$R \subset X \times X$$

**Equivalence Relation**

- Reflexive ()
- Symmetric ()
- Transitive ($x$ R $y$ and $y$ R $z$ then $x$ R $z$)

## 2.2 Logical Operators

**Negation**

| $p$ | $\neg p$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

**Conjuction :** Logical And

| $p$ | $q$ | $p \wedge q$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Disjuction :** Logical Or

| $p$ | $q$ | $p \vee q$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**Implication :** If $p$ Then $q$

| $p$ | $q$ | $p \implies q$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Equivalence :** $p$ if and only if $q$

| $p$ | $q$ | $p \iff q$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**For All :** $\forall$

**There Exists :** $\exists$

## 2.3 Functions

From pages 7-8 in *Introduction to the Theory of Computation* by Michael Sipser

**Function :** An objects that sets up an input-output relationship
**Domain :** The set of possible inputs to a function
**Range :** The set of possible outputs to a function

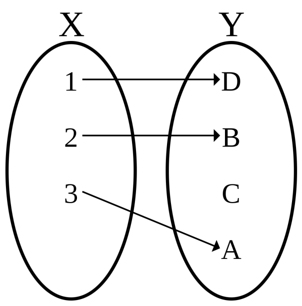
$f : D \to R$ Function $f$ has domain $D$ and range $R$

**One-To-One (Injection) :** If $x_1 \neq x_2$ then $f(x_1) \neq f(x_2)$

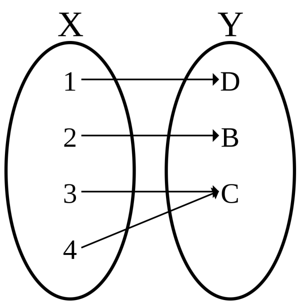
**Onto (Surjection) :** $f(X) = Y$

**Bijection :** One-To-One and Onto

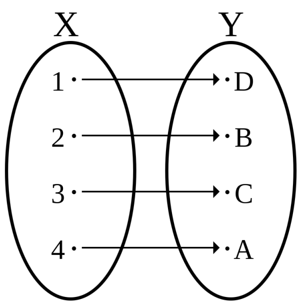
**Injection**



**Surjection**



**Bijection**



## 2.4 Logarithms

$$
\begin{aligned}
\log_b(XY) &= \log_b(X) + \log_b(Y) \\
\log_b\left(\frac{X}{Y}\right) &= \log_b(X) - \log_b(Y) \\
\log_b(X^y) &= y\log_b(X) \\
a^{\log_b(c)} &= c^{\log_b(a)}
\end{aligned}
$$

## 2.5 Summations

*From CLRS Appendix A*

**REMEMBER :** Summations are inclusive

Constants can be "taken out":

$$\sum_{i=1}^{n} cx_i = c\sum_{i=1}^{n} x_i$$

Addition can be broken up:

$$\sum_{i=1}^{n}(x_i + y_i) = \sum_{i=1}^{n} x_i + \sum_{i=1}^{n} y_i$$

**Arithmetic Series :**

$$\sum_{i=1}^{n} i = 1 + 2 + ... + n$$
$$\sum_{i=1}^{n} i = \frac{1}{2}n(n+1)$$
$$\sum_{i=1}^{n} i \in \Theta(n^2)$$

**Sum of Squares :**

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

**Sum of Cubes :**

$$\sum_{i=0}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$

**Geometric Series :** When $x \neq 1$ and is real

$$\sum_{i=0}^{n} x^i = 1 + x + x^2 + ... + x^n$$
$$\sum_{i=0}^{n} x^i = \frac{x^{n+1} - 1}{x - 1}$$

**Geometric Series :** When the summation is infinite and $|x| < 1$

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$$

**Harmonic Series :**

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}$$
$$H(n) = \sum_{i=1}^{n} \frac{1}{i}$$
$$H(n) = \ln n + O(1)$$

**Logarithms :**

$$S(n) = \sum_{i=1}^{n} \log(i)$$
$$S(n) = \log(1) + \log(2) + ... + \log(n-1) + \log(n)$$
$$S(n) = \log(1 * 2 * ... * (n-1) * n)$$
$$S(n) = \log(n!)$$

## 2.6 Asymptotic Notation

### 2.6.1 Table of Informal Definitions

The "Kinda like Saying" is entirely correct, it's just to wrap my head around the bounds and relations

| Name | Symbol | Informal Definition | Kinda like Saying |
|---|---|---|---|
| Little Omega | $\omega$ | Lower bound | $g(n) \in \omega(f(n))$ so $g(n) > f(n)$ |
| Big Omega | $\Omega$ | Tight Lower bound | $g(n) \in \Omega(f(n))$ so $g(n) \geq f(n)$ |
| Big Theta | $\Theta$ | Both an upper and lower bound | $g(n) \in \Theta(f(n))$ so $g(n) = f(n)$ |
| Big Oh | $O$ | Tight Upper bound | $g(n) \in O(f(n))$ so $g(n) \leq f(n)$ |
| Little Oh | $o$ | Upper bound | $g(n) \in o(f(n))$ so $g(n) < f(n)$ |

### 2.6.2 Table of Formal Definitions

| Name | Symbol | Formal Definition |
|---|---|---|
| Little Omega | $\omega$ | $g(n) \in \omega(f(n)) \iff \exists c > 0, n_0 > 0 \ni g(n) > cf(n) \forall n > n_0$ |
| Big Omega | $\Omega$ | $g(n) \in \Omega(f(n)) \iff \exists c > 0, n_0 > 0 \ni g(n) \geq cf(n) \forall n > n_0$ |
| Big Theta | $\Theta$ | $g(n) \in \Theta(f(n)) \iff \exists c_1 > 0, c_2 > 0, n_0 > 0 \ni c_1 \leq g(n) \leq c_2 f(n) \forall n > n_0$ |
| Big Oh | $O$ | $g(n) \in O(f(n)) \iff \exists c > 0, n_0 > 0 \ni g(n) \leq cf(n) \forall n > n_0$ |
| Little Oh | $o$ | $g(n) \in o(f(n)) \iff \exists c > 0, n_0 > 0 \ni g(n) < cf(n) \forall n > n_0$ |

### 2.6.3 Table of Limit Definitions

| Name | Symbol | Proving with Limits |
|---|---|---|
| Little Omega | $\omega$ | $\lim_{n \to \infty} f(n)/g(n) = \infty$ |
| Big Omega | $\Omega$ | $\lim_{n \to \infty} f(n)/g(n) \neq 0$ |
| Big Theta | $\Theta$ | $\lim_{n \to \infty} f(n)/g(n) \neq 0, \infty$ |
| Big Oh | $O$ | $\lim_{n \to \infty} f(n)/g(n) \neq \infty$ |
| Little Oh | $o$ | $\lim_{n \to \infty} f(n)/g(n) = 0$ |

### 2.6.4 Properties

**Transitivity :**

- $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$ : $f(n) \in O(h(n))$
- $f(n) \in \Omega(g(n))$ and $g(n) \in \Omega(h(n))$ : $f(n) \in \Omega(h(n))$

**Reflexivity :** $f(n) \in \Theta(f(n))$

**Transpose Symmetry :** $f(n) \in O(g(n) \iff g(n) \in \Omega(f(n))$

**Symmetry :** $f(n) \in \Theta(g(n)) \iff g(n) \in \Theta(f(n))$

**Trichotomy :** For any two real numbers $a$ and $b$: $a > b$ or $a = b$ or $a < b$

# 3 Proofs

From pages 17-23 in *Introduction to the Theory of Computation* by Michael Sipser

**Definition :** Describes and object or notation
**Proof :** Convincing logical argument that a statement is true
**Theorem :** A mathematical statement proved true
**Lemma :** A theorem that assists in the proof of another theorem
**Corollaries :** The parts of a theorem where we can conclude that other related statements are true

## 3.1 Proof By Construction

To prove something exists, we prove we can construct the object in a general case.

## 3.2 Proof By Contradiction

In order to prove a theorem true, we can say it's false, then show that that leads to an even more false statement. By disproving the false statement, we show the contradiction is true.

## 3.3 Proof By Induction

In induction, we need an ordered set of variables with consistent variance. We create a base case, prove it is true, then for the inductive case, we prove that taking one step forward is also true and reinforces the base case

Prove:

$$S(n) = \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$\begin{aligned}
S(n) &= \sum_{i=0}^{n} 2^i && \\
S(0) &= \sum_{i=0}^{0} 2^i && \text{Base Case} \\
S(0) &= 1 && \\
2^{0+1} - 1 &= 1 && \\
S(n) &\rightarrow S(n+1) && \text{Inductive Case} \\
S(n+1) &= \sum_{i=0}^{n+1} 2^i && \\
S(n+1) &= \sum_{i=0}^{n} 2^i + 2^{n+1} && \\
S(n+1) &= 2^{n+1} - 1 + 2^{n+1} && \\
S(n+1) &= 2^{n+2} - 1 && \\
S(n+1) &= 2^{(n+1)+1} - 1 && \text{Induction Holds}
\end{aligned}$$

## 3.4 Proof By Structural Induction

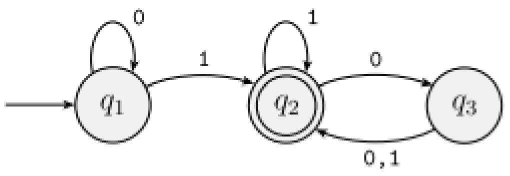
Structural induction is similar to regular induction, except instead of using a base case and a literal $n + 1$ case, it's analyzing the data structure to see what a logical "plus one" step would be. This can be adding subtrees, edges, vertices, etc.

# 4 Finite Automata

## 4.1 Deterministic Finite Automata

From pages 31-43 in *Introduction to the Theory of Computation* by Michael Sipser

A finite automaton, or finite state machine, is the simplest way to describe a computational models behavior.



In the above state machine, there are three states denoted by circles, $q_1$, $q_2$, and $q_3$. The transitions are denoted by arrows. The start state is denoted by the arrow coming from no where. The accept state is denoted by the bubble with a circle inside it. Upon receiving a string, it processes it then produces an "accept" or "reject" output. Only if we end in an accept state is the output "accept".

$A$ is the set of all strings that the state machine accepts. $\Sigma$ is the alphabet of symbols that a string, $w$, is composed of. $A$ is the language of the machine where $L(M) = A$. M recognizes A. M accepts A.

While the simple description of a finite automaton is simple enough for me to understand, there are five things a finite automaton must include. This can be described using the 5-tuple, $(Q, \Sigma, \delta, q_0, F)$.

1. **States :** A set of states ($Q$)
2. **Alphabet :** The allowed input symbols ($\Sigma$)
3. **Transition Function :** The rules for moving from one state to another ($\delta : Q \times \Sigma \to Q$)
4. **Start State :** Self Explanatory ($q_0 \in Q$)
5. **Set of Accept States :** Self explanatory ($F \subseteq Q$)

Looking back at the above state machine, we can describe it as the following.

1. $Q = \{q_1, q_2, q_3\}$
2. $\Sigma = \{0, 1\}$
3. $\delta$ can be described in the following table with states on the left, input on the top, and the state transitioned to being the intersection

| | 0 | 1 |
|---|---|---|
| $q_1$ | $q_1$ | $q_2$ |
| $q_2$ | $q_3$ | $q_2$ |
| $q_3$ | $q_2$ | $q_2$ |

4. $q_0 = q_1$
5. $F = \{q_2\}$ (in this case, a singleton set)

## 4.2 Non-deterministic Finite Automata

From pages 47-50 and 53 in *Introduction to the Theory of Computation* by Michael Sipser

**Deterministic :** Given a state and an input symbol, the next state is determined by a function and we know what it will be.

**Non-Deterministic :** There are many options in a given state.

**Non Deterministic Finite Automata (NFA)**

- Can have zero, one, or many transitions for each alphabet member
- Can include $\varepsilon$ in addition to the alphabet
- Traverses in a tree

**Deterministic Finite Automata (DFA)**

- Each state has one transition for each alphabet member
- Only includes alphabet members
- Traverses sequentially
- All DFAs are NFAs

An NFA is determined with a reject/accept state in a different way than a DFA. A DFA traverses sequentially, one state after the next, an NFA traverses and creates a tree of all the possible outcomes. Some rules when traversing:

- If there is not a transition for an input, the branch stops
- If there are multiple transitions for an input, it branches for as many times with than input.
- Upon reaching $\varepsilon$

We also have a more formal definition of an NFA:

1. **States :** A finite set of states ($Q$)
2. **Alphabet :** The finite allowed input symbols ($\Sigma$)
3. **Transition Function :** The rules for moving from one state to another ($\delta : Q \times \Sigma_\varepsilon \to P(Q)$) $P(Q)$ is the power set of $Q$
4. **Start State :** Self Explanatory ($q_0 \in Q$)
5. **Set of Accept States :** Self explanatory ($F \subseteq Q$)

### 4.2.1 Subset Construction

**Theorem :** Subset Construction

Let an NFA $N$ be given (we assume for the sake of simplicity that $N$ doesn't have any $\varepsilon$ transitions), we can construct an equivalent DFA $D$ where the language of $D$ is the same as the language of $N$ ($L(D) = L(N)$).

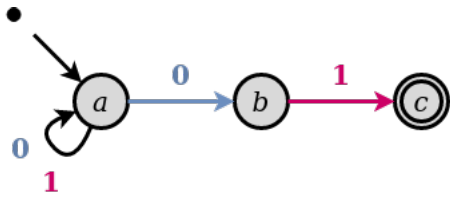
$$\begin{aligned} N &= (Q, \Sigma, \delta, q_0, F) \\ D &= (Q', \Sigma, \delta', q_0', F') \end{aligned}$$

Where $Q'$, $q_0'$, $\delta'$, and $F'$ are defined as:

$$\begin{aligned} Q' &= P(Q) && \text{the set consisting of all subsets of } Q \\ q_0' &= \{q_0\} \\ \delta'(R, a) &= \bigcup_{r \in R} \delta(r, a) \\ F' &= \{R \in Q' \mid R \cap F \neq \varnothing\} \end{aligned}$$

**Example:**

Given the NFA for a language where all strings end in "01":



$$Q = \{a, b, c\}$$
$$\Sigma = \{0, 1\}$$
$$\delta = \begin{array}{c|cc} & 0 & 1 \\ \hline a & \{a,b\} & \{a\} \\ b & \varnothing & \{c\} \\ c & \varnothing & \varnothing \end{array}$$
$$q_0 = a$$
$$F = \{c\}$$

To turn this into a DFA, we first find the states $Q'$ which is a powerset of $Q$. *Note: I concatenated the letters for the state names instead of making a set of sets for cleanliness and readability.*

$$Q' = \{\varnothing, a, b, c, ab, ac, bc, abc\}$$

Then our starting state $q_0'$ stays the same, only now represented as a set:

$$q_0' = \{a\}$$

$\Sigma$ doesn't change, so we can move onto $\delta'$. Let's start with a blank table:

$\delta' =$

| | 0 | 1 |
|---|---|---|
| $\varnothing$ | | |
| $a$ | | |
| $b$ | | |
| $c$ | | |
| $ab$ | | |
| $ac$ | | |
| $bc$ | | |
| $abc$ | | |

Now let's break down the big union operation:

$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a)$$

It's saying for each state $R$ in the DFA (which is a set of states) and input character $a$, the state we transition to is the union of all the transitions from the substates in the NFA. So $\delta'(abc, 0)$ transitions to the union of $\delta(a, 0)$, $\delta(b, 0)$, and $\delta(c, 0)$. This will give a set we transition to, but remember that each state in the DFA, $R$, is a set. This gives the following $\delta'$:

$\delta' =$

| | 0 | 1 |
|---|---|---|
| $\varnothing$ | $\varnothing$ | $\varnothing$ |
| $a$ | $ab$ | $a$ |
| $b$ | $\varnothing$ | $c$ |
| $c$ | $\varnothing$ | $\varnothing$ |
| $ab$ | $ab$ | $ac$ |
| $ac$ | $ab$ | $a$ |
| $bc$ | $\varnothing$ | $c$ |
| $abc$ | $ab$ | $ac$ |

We'll now notice that many states are unreachable. No state ever transitions to $abc$, so it's transitions aren't necessary and we can remove it from the diagram. We then get:

$\delta' =$

| | 0 | 1 |
|---|---|---|
| $\varnothing$ | $\varnothing$ | $\varnothing$ |
| $a$ | $ab$ | $a$ |
| $b$ | $\varnothing$ | $c$ |
| $c$ | $\varnothing$ | $\varnothing$ |
| $ab$ | $ab$ | $ac$ |
| $ac$ | $ab$ | $a$ |
| $bc$ | $\varnothing$ | $c$ |
| $abc$ | $ab$ | $ac$ |

$\rightarrow$

| | 0 | 1 |
|---|---|---|
| $\varnothing$ | $\varnothing$ | $\varnothing$ |
| $a$ | $ab$ | $a$ |
| $c$ | $\varnothing$ | $\varnothing$ |
| $ab$ | $ab$ | $ac$ |
| $ac$ | $ab$ | $a$ |

$\rightarrow$

| | 0 | 1 |
|---|---|---|
| $\varnothing$ | $\varnothing$ | $\varnothing$ |
| $a$ | $ab$ | $a$ |
| $ab$ | $ab$ | $ac$ |
| $ac$ | $ab$ | $a$ |

$\rightarrow$

| | 0 | 1 |
|---|---|---|
| $a$ | $ab$ | $a$ |
| $ab$ | $ab$ | $ac$ |
| $ac$ | $ab$ | $a$ |

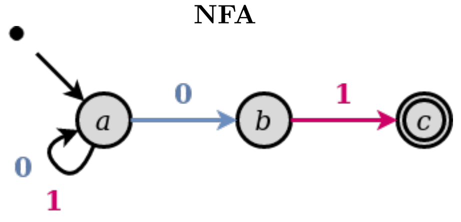
$$Q' = \{a, ab, ac\}$$

Our finishing state set $F'$ is the set states in $Q'$ that when unioned with the original finishing set $F$ isn't empty. The finishing set was $F = \{c\}$, but $c \notin Q'$. However, $ac \in Q'$ which contains $c$ and is a member of the new finishing set:

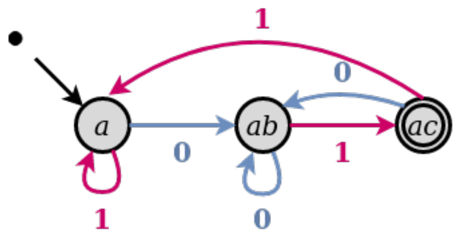
$$F' = \{ac\}$$

Putting it all together, side by side we get:

**NFA**



**DFA**

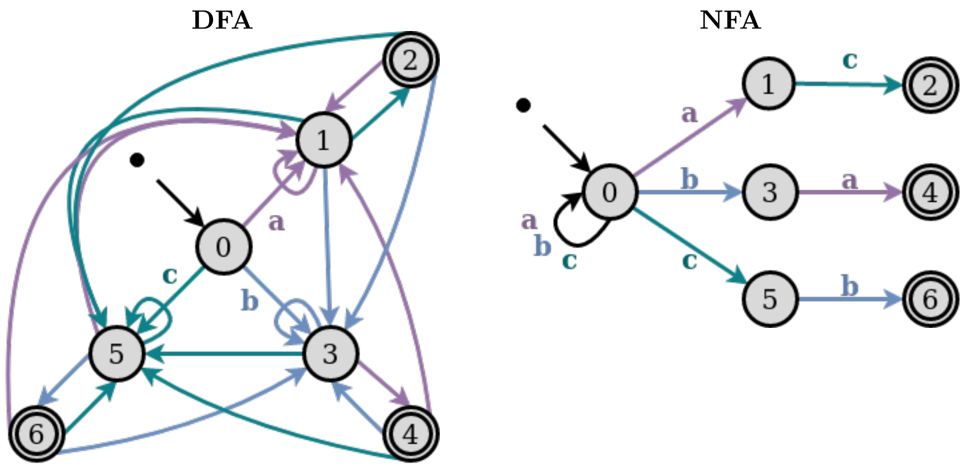


$$Q = \{a, b, c\}$$
$$\Sigma = \{0, 1\}$$
$$\delta = \begin{array}{c|cc} & 0 & 1 \\ \hline a & \{a, b\} & \{a\} \\ b & \varnothing & \{c\} \\ c & \varnothing & \varnothing \end{array}$$
$$q_0 = a$$
$$F = \{c\}$$

$$Q' = \{a, ab, ac\}$$
$$\Sigma = \{0, 1\}$$
$$\delta' = \begin{array}{c|cc} & 0 & 1 \\ \hline a & ab & a \\ ab & ab & ac \\ ac & ab & a \end{array}$$
$$q_0' = a$$
$$F = \{ac\}$$

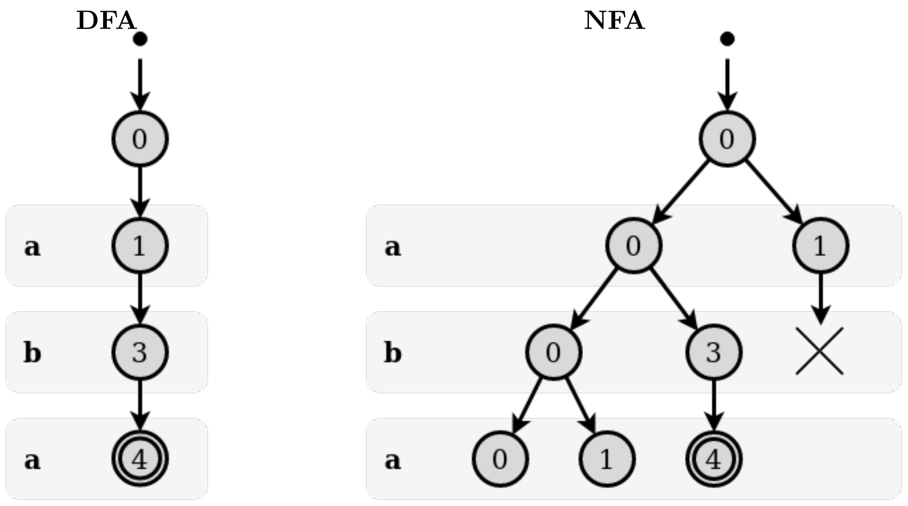
I did this before looking at the subset construction, but here's another, more complex, example showing the traversal of a DFA and NFA.

To more visually show the differences between equivalent DFAs and NFAs, consider the language that accepts all strings ending in "ac", "ba", or "cb". Since the DFA transitions got messy, I color coded them. All **"a"** transitions are **purple**, **"b"** transitions are **blue**, and **"c"** transitions are **teal**.



The NFA is a lot easier to read because for the most part, it only requires the "necessary" transitions that the DFA has. The DFA needs to have all possible transitions for each state.

The traversal of each given the string "aba":



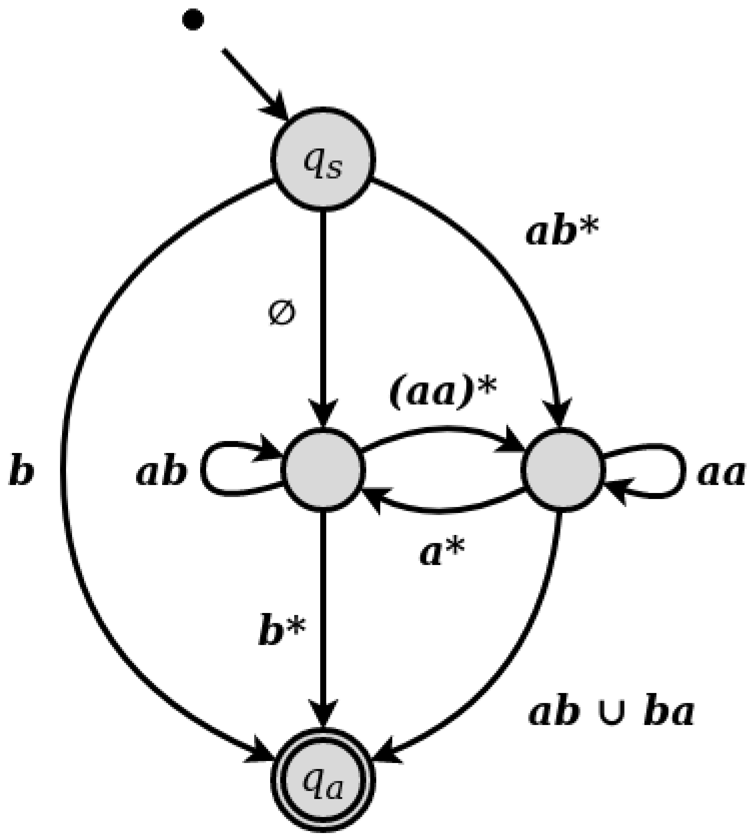
## 4.3 Generalized Non-Deterministic Finite Automata

A GNFA is an NFA that reads multiple input symbols at a time using regular expressions to connect states. GNFAs have a few extra properties that aren't required by NFAs:

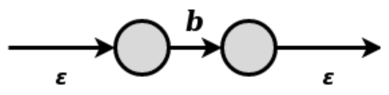
- The state state has transition arrows going to every other state and no arrows coming in from any other state
- There is only one accept state. It is pointed to by every other state and points to no states
- Every other state has an arrow to every other state and itself

The formal definition of a GNFA is a 5-tuple $(Q, \Sigma, \delta, q_{start}, q_{accept})$:

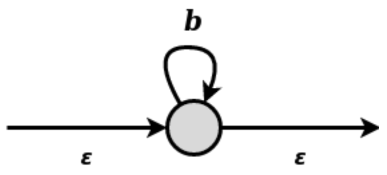
- $Q$ : The finite set of states
- $\Sigma$ : The input alphabet
- $\delta : (Q - \{q_{accept}\}) \times (Q - \{q_{start}\}) \rightarrow \mathcal{R}$ : The transition function
- $q_{start}$ : The start state
- $q_{accept}$ : The accept state



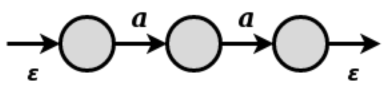
The reason we can use regular expressions as transitions is because we can expand those regular expressions into small NFAs:



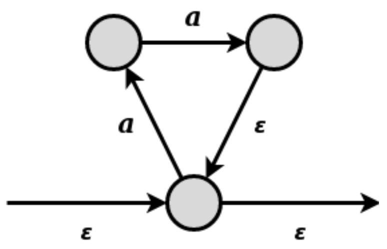
$\mathbf{b}$



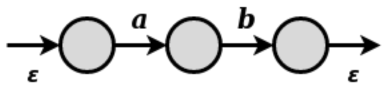
$\mathbf{b}^*$



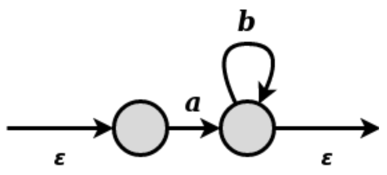
$\mathbf{aa}$



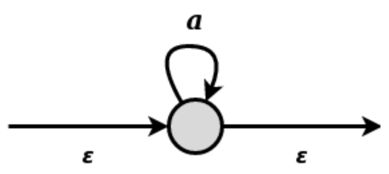
$(\mathbf{aa})^*$



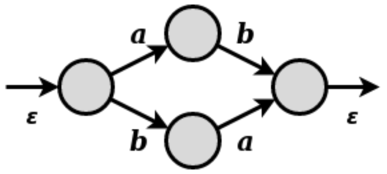
$\mathbf{ab}$



$\mathbf{ab}^*$

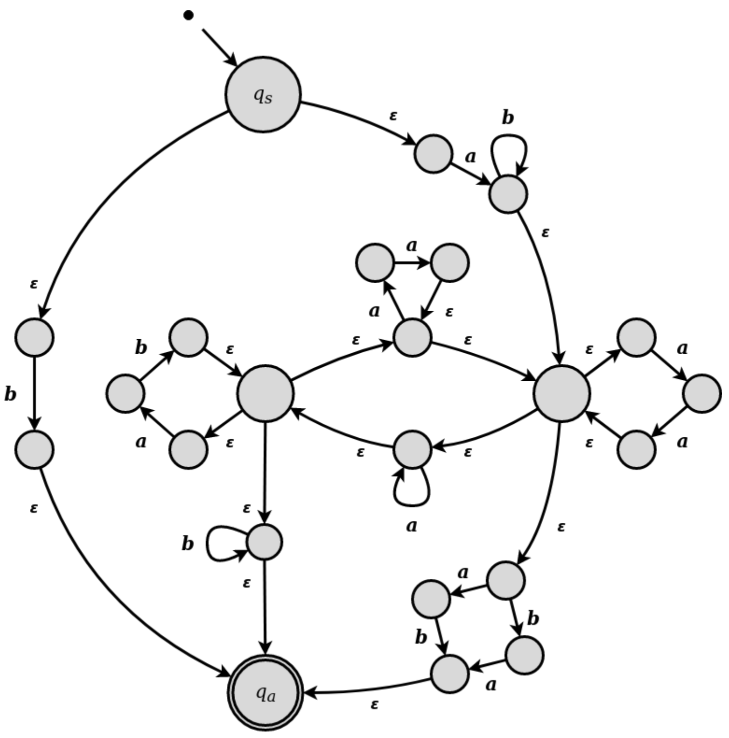


$\mathbf{a}^*$

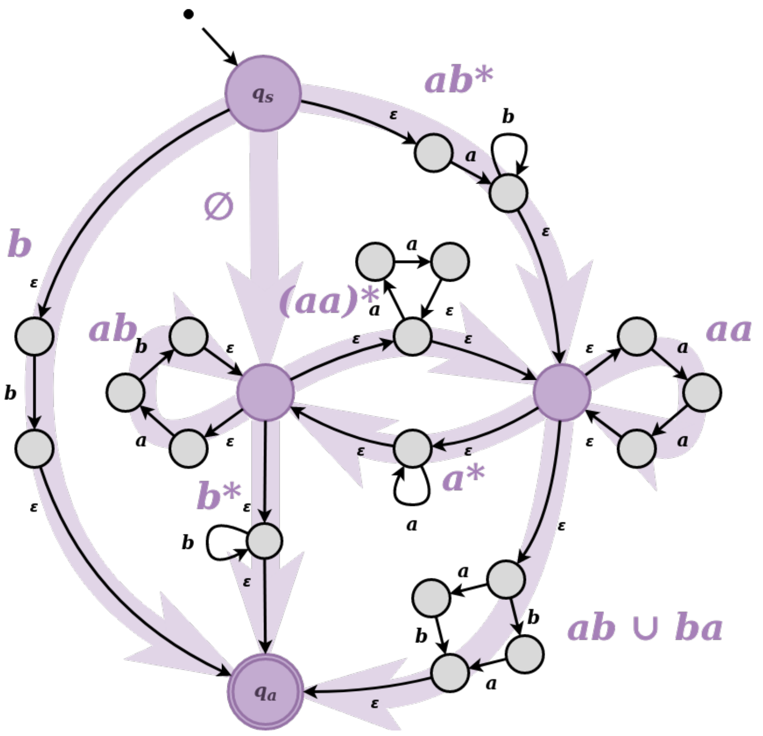


$\mathbf{ab} \cup \mathbf{ba}$

Which we can put back together to get:



Then again, with the GNFA highlighted in purple for clarity:



### 4.3.1 DFAs to GNFAs to Regex

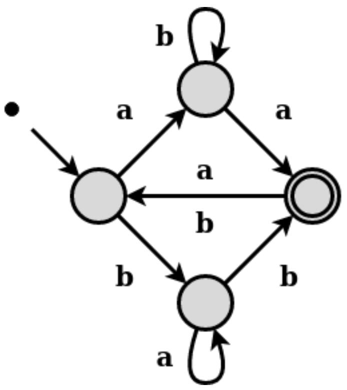
To get a regular expression from a DFA, we can:

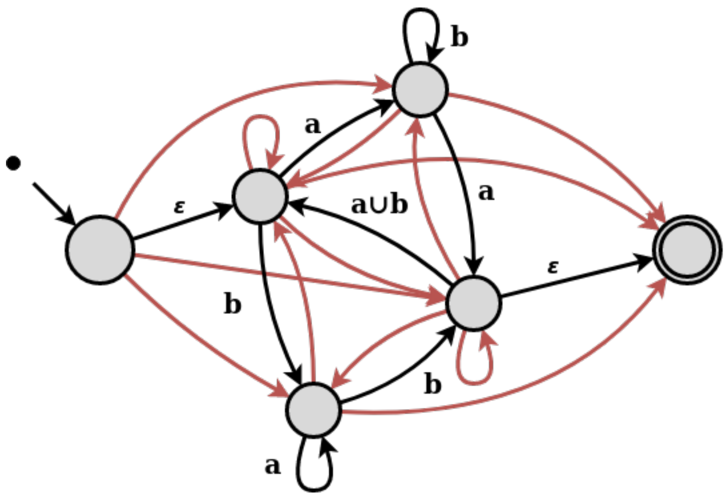
1. Convert the DFA into a GNFA
2. Combine states in the GNFA using regex until two states and one transition remains

Coverting a DFA into a GNFA is easy, we need to:

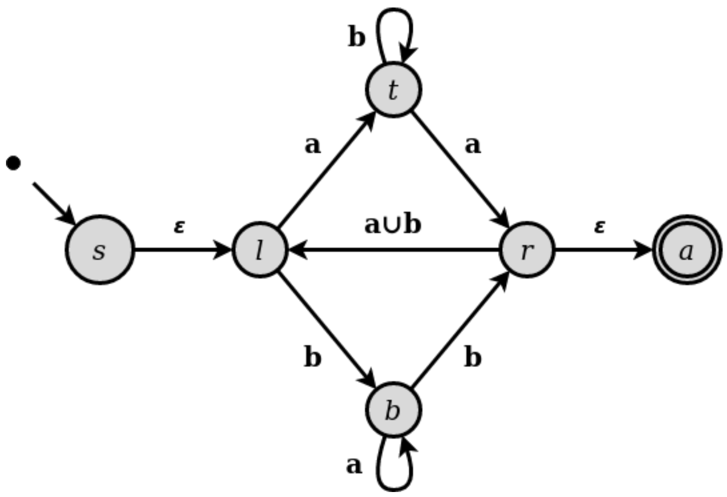
1. Create a new start state with a $\varepsilon$ transition to the old start state
2. Create a new single accept state with $\varepsilon$ transistions from the old accept states
3. If any states have multiple labels or multiple arrows between the same states in the same direction, union them together into one arrow
4. Between states that had no arrows, add an $\varnothing$ transistion

Apply this and we'll get the following:



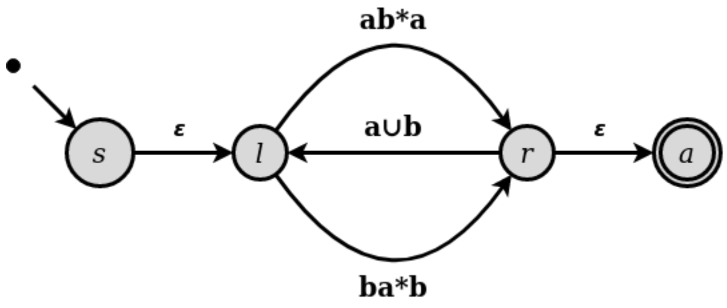


Above, the **red** arrows are the $\varnothing$ transistions. I included them for the sake of following instructions, but these transistions cannot be travelled, so I can remove them to get a much cleaner diagram that I've also given labels to assist in the next step:

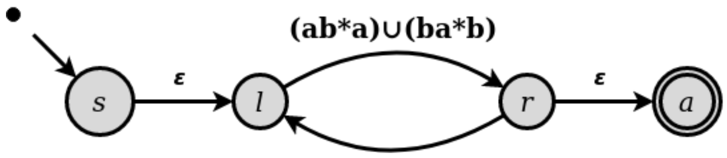


Now that we have our GNFA, we want to combine states using regular expressions to get to the point where we only have a start and accept state.

We can start by removing these top $t$ and bottom $b$ states. There's a single transition to each (we'll call it $R_i$), a single transistion out ($R_o$), and a repeated transistion to itself ($R_r$). This can be simplified to the regular expression $R_i R_r^* R_o$ and we can replace those states:



We then see that there are two transistions from the "left state" $l$ to the "right state" $r$. We can union these together and get:



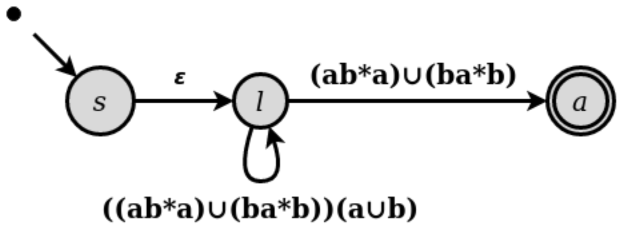
This next part is a little wonky. Now when we replace a state, we need to change the transistions from the start state or to the accept state.

If we were to just remove $r$ and "collapse" the $\varepsilon$ transistion, we'd break the GNFA structure because there would be an exit transistion from the accept state $a$ to $l$, which isn't allowed.

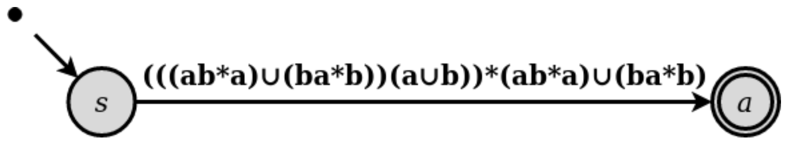
I'm going to first define $R_1 = (ab^*a) \cup (ba * b)$ (the transistion from $l$ to $r$). $R_1$ can point directly to $a$ without any issues since we're just concatenating it with the empty string ($\varepsilon$).

Then I'm going to define $R_2 = a \cup b$ (the transistion from $r$ to $l$). As said before, $R_2$ cannot point from $a$ to $l$. In this case though, we see that a "loop" is formed. Through some analysis, I see that I can create a transistion from $l$ to $l$ (itself) by traveling down $R_1$ then coming back through $R_2$. I can create a new transistion on $l$ by concatenating $R_1$ and $R_2$.

This all creates:



Finally, to replace $l$, we can apply the same property we did first with $R_i = \varepsilon$, $R_r = ((ab^*a) \cup (ba^*b))(a \cup b)$, and $R_o = (ab^*a) \cup (ba^*b)$ to get quite possibly the most hideous regular expression I've ever written:



## 4.4 Minimal DFAs

The MINIMIZE algorithm (page 299 of Sipser) takes a DFA $M = (Q, \Sigma, \delta, q_0, A)$ as input:

1. Remove all states $M$ that are unreachable from the start state
2. Construct the following undirected graph $G$ whose nodes are the states of $M$
3. Place an edge in $G$ connecting every accept state with every non-accept states. Additional edges are added as follows:
4. Repeat until no new edges are added to $G$
   (a) For every pair of distinct states $q$ and $r$ of $M$ and every $a \in \Sigma$
      - Add the edge $(q, r)$ to $G$ if $(\delta(q, a), \delta(r, a))$ is an edge of $G$
5. For each state $q$, let $[q]$ be the collection of states $[q] = \{r \in Q \mid \text{no edge joins } q \text{ and } r \text{ in } G\}$
6. Form a new DFA $M' = (Q', \Sigma, \delta', q_0', A')$ where:
   - $Q' = \{[q] \mid q \in Q\}$ (if $[q] = [r]$, only one is in $Q'$)
   - $\delta'([q], a) = [\delta(q, a)]$ for every $q \in Q$ and $a \in \Sigma$
   - $q_0' = [q_0]$
   - $A' = \{[q] \mid q \in A\}$
7. Return $M'$

# 5 Regular Languages

From pages 44-45 in *Introduction to the Theory of Computation* by Michael Sipser

**Unary Operator :** An operation on one element

**Binary Operator :** An operation on two elements

**Regular Language :** A language that can be expressed with a regular expression

## 5.1 Operations

**Union :** $A \cup B = \{x|x \in A \text{ or } x \in B\}$

**Intersection :** $A \cap B = \{x|x \in A \text{ and } x \in B\}$

**Concatenation :** $A \circ B = \{xy|x \in A \text{ and } y \in B\}$

Concatenation is like a cross product but concatenating the tuples.

**Complement :** $\bar{L}$ is the opposite of $L$ in $\Sigma^*$

**Star :** $A^* = \{x_1x_2...x_k|k \geq 0 \text{ and } x_i \in A\}$

In star, $\varepsilon$ (the empty string) is always a member of the language. This is because the star means 0 or more times.

Given alphabet $\Sigma$ where it's the standard English alphabet, a through z. If $A = \{\text{good}, \text{bad}\}$ and $B = \{\text{girl}, \text{boy}\}$, then

$$\begin{aligned}
A \cup B &= \{\text{good}, \text{bad}, \text{boy}, \text{girl}\} \\
A \circ B &= \{\text{goodgirl}, \text{goodboy}, \text{badgirl}, \text{badboy}\} \\
A^* &= \{\varepsilon, \text{good}, \text{bad}, \text{goodgood}, \text{goodbad}, \text{badgood}, \text{badbad}, \text{goodgoodgood...}\}
\end{aligned}$$

## 5.2 Properties

- A language $L \in \Sigma^*$ is considered regular is $L = L(A)$ for some DFA $A$
- A language $L \in \Sigma^*$ is considered regular is $L = L(B)$ for some NFA $B$
- If $L_1$ and $L_2$ are regular then so are the languages described by $L_1 \cap L_2$ and $L_1 \cup L_2$. Regular languages are closed under union and intersection
- If $L \subseteq \Sigma^*$ is regular, so is $\bar{L}$. This is achieved by inverting the "accept" and "deny" states in a DFA
- If a language can be described by a regular expression, then it is regular

#### Theorem

1.25 in *Introduction to the Theory of Computation* by Michael Sipser

The class of regular languages is closed under the union operation. In dumb bitch english, if $A_1$ is regular and $A_2$ is regular, then $A_1 \cup A_2$ is regular.

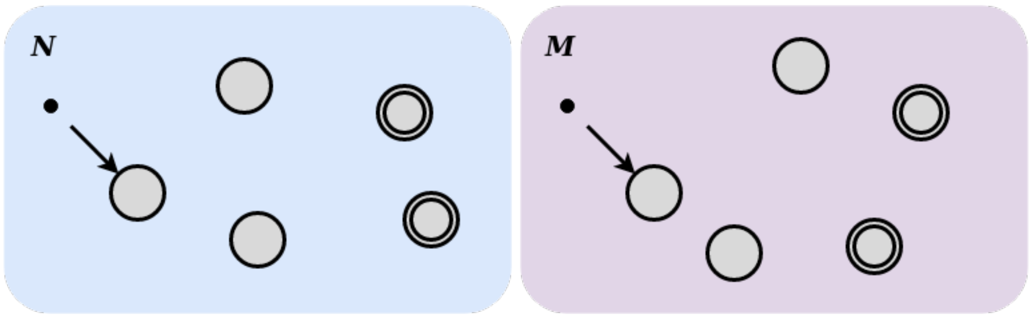
To prove this we could make two machines $M_1$ and $M_2$ that both separately accept $A_1$ and $A_2$ respectively. We then could combine them into a machine $M$, however the machine would not know the difference between something $M_1$ should accept vs $M_2$ should accept and could confuse inputs.

Given $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ accepts $A_1$, $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ accepts $A_2$, construct an NFA $N = (Q, \Sigma, \delta, q_0, F)$ accepts $A_1 \cup A_2$.

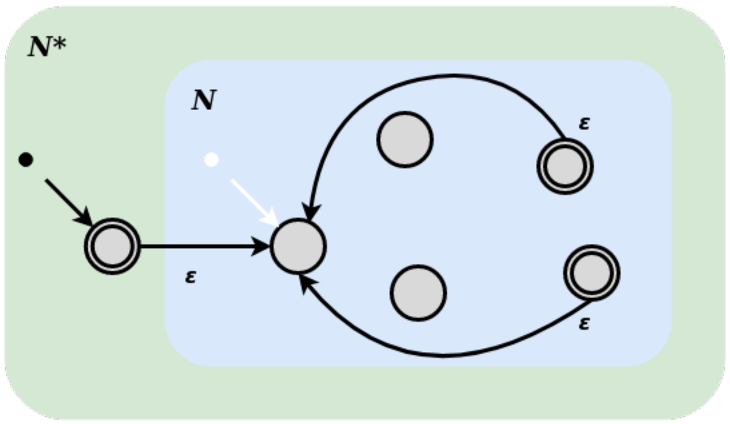
1. $q_0$ is the start state of $N$. It's new
2. $Q = \{q_0\} \cup Q_1 \cup Q_2$
3. $F = F_1 \cup F_2$
4. Define $\delta$ for any $q \in Q$ and for any $a \in \Sigma_\varepsilon$ as:

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \\ \delta_2(q,a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \text{ and } a = \varepsilon \\ \varnothing & q = q_0 \text{ and } a \neq \varepsilon \end{cases}$$

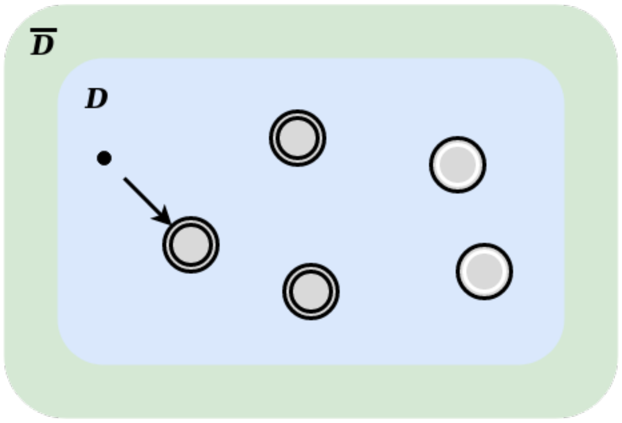
To more visually show closure under these operations, say we the NFAs $N$ and $M$:



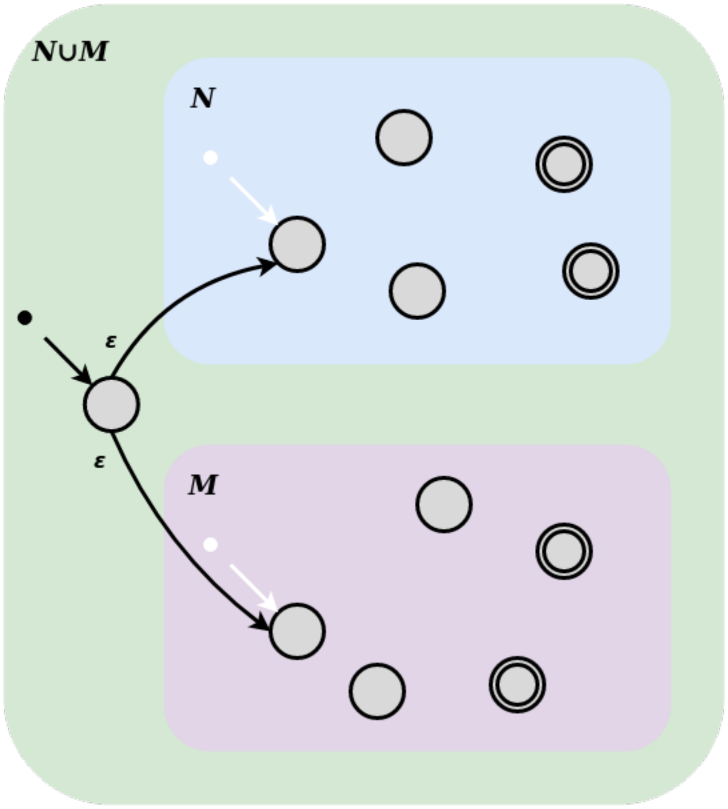
**Star :** Create a new start state that accepts the empty string and have a single $\varepsilon$ transition to the former start state. Add $\varepsilon$ transitions from each accept state to the former state state. To get the corresponding DFA, use subset construction.



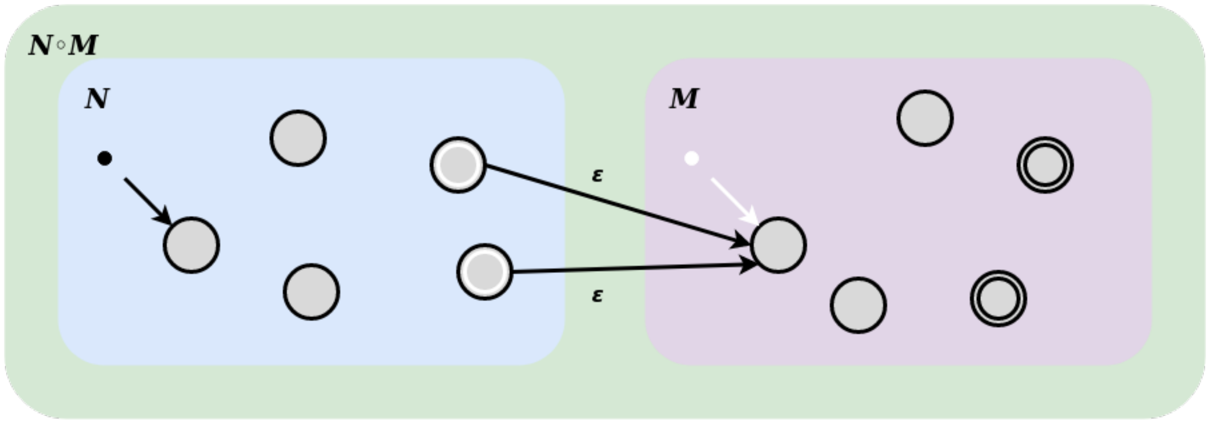
**Complement :** Swap the accept and not accept states. This only applies to DFAs



**Union :** Create a new start state with $\varepsilon$ transitions to the former start states of $N$ and $M$. Again, to get the DFA you can use subset construction



**Concatenation :** Change all of $N$'s accept states to be non-accepting. Create a $\varepsilon$ transition from each of these to $M$'s start state. Remove $M$'s start state. This creates one machine with $N$'s start state and $M$'s accepting states. Again, to get the DFA, use subset construction



**Closure :** Given a language $L$, $L^*$ is a closure of $L$ if $L^* = \{w = v_1 v_2 \dots v_l \mid v_i \in L, i = 1, 2, \dots, k, k \geq 1\}$

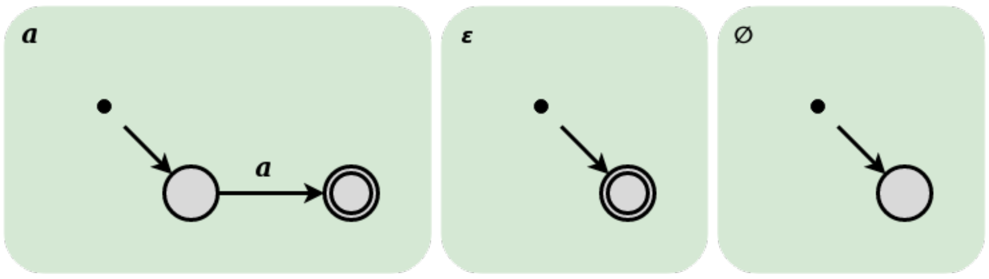
## 5.3 Regular Expressions

This is like regex in UNIX but a lot simpler. Below is the formal definition of a regular expression:

1. $a$ for some $a$ in the alphabet $\Sigma$
2. $\varepsilon$
3. $\varnothing$
4. $(R_1 \cup R_2)$, where $R_1$ and $R_2$ are regular expressions
5. $(R_1 \circ R_2)$ or $(R_1 R_2)$, where $R_1$ and $R_2$ are regular expressions
6. $(R_1^*)$, where $R_1$ is a regular expression

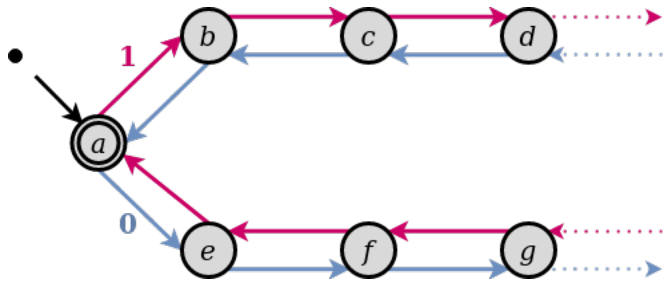
For example, $(0 \cup 1)^*$ is a sequence of as many 0s and 1s

Above, we've already seen the proofs for $(R_1 \cup R_2)$, $(R_1 R_2)$, and $R_1^*$ in NFAs. The NFAs for $a$, $\varepsilon$, and $\varnothing$ are below:



## 5.4 Non Regular Languages

A language that isn't regular. It cannot be constructed with a DFA or NFA. An example of one is a language with an equal number of "0"s and "1"s. Since there can be an infinite number of "0"s and "1"s, there would have to be an infinite number of states to account for infinite possibilities where there are more "0"s than "1"s and vice versa. This is shown in the incomplete example below. *Note: **1** transitions are shown in **pink** and **0** transitions are shown in **blue***



State $a$ is the only accept state where there are equal "0"s and "1"s. When a "1" is encountered, we transition to state $b$ to show there's one more "1" than "0". If a "0" were encountered, we'd move back to $a$ to show an equal number of "0"s and "1"s. If instead, a "1" were encountered, we'd move to state $c$ to show there are two more "1"s than "0"s. If a "0" were then encountered, we'd move back to $b$. Since there can be an infinite number of "0"s and "1"s, there can be infinite possible of differences in counts. Since we'd need infinite states to represent this, we cannot build a *finite* automata to describe this language, proving it's not regular.

## 5.5 Pumping Lemma

All strings in a regular language have a special property, if the language does not have this property, it is not regular. The property is:

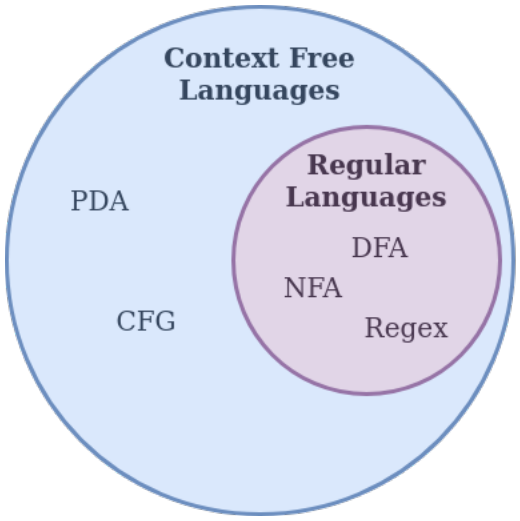
For a regular language $A$, there is a number $p$ (the pumping length) where if $s$ is any string in $A$ of at least length $p$, then $s$ can be divided into at least 3 pieces, where $s = xyz$, where the following conditions are met.

1. for each $i \geq 0, xy^i z \in A$
2. $|y| > 0$, and
3. $|xy| \leq p$

# 6 Context Free Languages

We're starting to work our way into larger "sets of computing" or hierarchies of computing.

We started with regular languages which were able to be computed through DFAs and NFAs and regular expressions. Regular languages are a subset of context free languages, which can be computed with a Pushdown Automata (PDA) or a context free grammars (CFG). All regular languages are context free. Not all context free languages are regular.

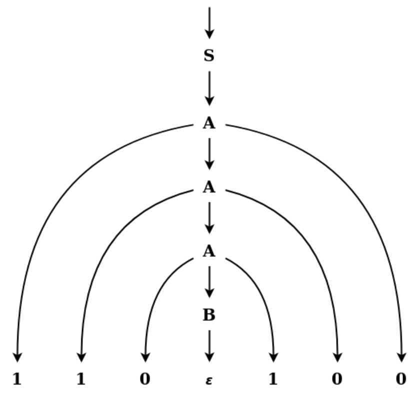


## 6.1 Context Free Grammars

CFGs are to context free languages as regular expressions are to regular languages. They have sexy recursive structure which allows us to represent all regular languages with a CFG plus other non regular languages.

For example, we said earlier that the language $L = \{w \mid w \text{ has an equal number of 0s and 1s } \}$ is not regular and showed we could not produce a DFA to accept the string. $L$ is not regular, but it is context free and can be captured with the following CFG:

$$S \to A$$
$$A \to 0A1 \mid 1A0 \mid B$$
$$B \to \varepsilon$$



**Productions :** The set of substitution rules
**Variable :** The symbol on the left of a production
**Terminal :** The product of a production. A cobination of symbols and products
**Start Variable :** The start of a CFG

The formal definition of a CFG is a 4 tuple $G = (V, \Sigma, R, S)$ where:

1. $V$ is a finite set of variables
2. $\Sigma$ is a finite set, disjoint from $V$ of terminals
3. $R$ is a finite set of rules, each with the form $A \to w$ where $A \in V$ and $w \in (V \cup \Sigma)^*$
4. $S \in V$ is the start variable

**Yields :** $uAv \Rightarrow uwv$

- $u \in (V \cup \Sigma)^*$
- $v \in (V \cup \Sigma)^*$
- $A \to w$ is a production in $G$

**Derives :** $u \overset{*}{\Rightarrow} v$

- $u \in (V \cup \Sigma)^*$
- $v \in (V \cup \Sigma)^*$
- $u = v$ or $\exists u_1, u_2, \ldots, u_k$ where $u_i \in (V \cup \Sigma)^* \forall i \in [1, k]$ such that $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$

Context Free Languages are closed under union and concatenation. They are not closed under intersection.

Given a context free language $L_A$ recognized by a CFG $G_A = (V_A, \Sigma_A, R_A, S_A)$ and a context free language $L_B$ recognized by a CFG $G_B = (V_B, \Sigma_B, R_B, S_B)$, we can construct the following:

**Concatenation**

$$L = L_A L_B$$

We can construct a CFG $G = (V, \Sigma, R, S)$:

$$\begin{aligned} V &= V_A \cup V_B \\ \Sigma &= \Sigma_A \cup \Sigma_B \\ R &= R_A \cup R_B \\ S &\to S_A S_B \end{aligned}$$

**Union**

$$L = L_A \cup L_B$$

We can construct a CFG $G = (V, \Sigma, R, S)$:

$$\begin{aligned} V &= V_A \cup V_B \\ \Sigma &= \Sigma_A \cup \Sigma_B \\ R &= R_A \cup R_B \\ S &\to S_A \mid S_B \end{aligned}$$

### 6.1.1 Chomsky Normal Form

A CFG is in Chomsky Normal Form (CNF) if every rule has the form:

$$\begin{aligned} A &\rightarrow BC \\ A &\rightarrow a \end{aligned}$$

Where:

- $a$ is any terminal ($a \in \Sigma$)
- $A$, $B$, and $C$ are any variables ($A, B, C \in V$)
- $B$ and $C$ are not the start state ($B \neq S$) and ($C \neq S$)
- $S \rightarrow \varepsilon$ is allowed

To convert any CFG to Chomsky Normal Form:

1. Introduce a new start state $S_0 \rightarrow S$ where $S$ is the former start state
2. Eliminate all rules of the form $A \rightarrow \varepsilon$
3. Eliminate all unit rules of the form $A \rightarrow B$
4. Ensure the language is still accurate
5. Convert remaining rules to the proper form

*Example on pages 118 and 119 of Sipser.* Let's start with the following CFG:

$$\begin{aligned} S &\rightarrow ASA \mid aB \\ A &\rightarrow S \mid B \\ B &\rightarrow b \mid \varepsilon \end{aligned}$$

First, we introduce the new start variable $S_0$:

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow ASA \mid aB \\ A &\rightarrow S \mid B \\ B &\rightarrow b \mid \varepsilon \end{aligned}$$

Next, we remove the $\varepsilon$ rule $B \rightarrow \varepsilon$. To do this accurately, we need to find every production where $B$ is referenced and add a new rule to make the same production with $B$ replaced by $\varepsilon$.

For example, for $S \rightarrow aB$, we will add a rule $S \rightarrow a\varepsilon$ or just $S \rightarrow a$

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow ASA \mid aB \mid a \\ A &\rightarrow S \mid B \mid \varepsilon \\ B &\rightarrow b \end{aligned}$$

In doing this, we created a new $\varepsilon$ rule $A \rightarrow \varepsilon$. The process is the same, however this time the rule $S \rightarrow ASA$ contains two $A$s. This time, we need to generate each possible combination replacing $A$ with $\varepsilon$:

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow ASA \mid AS \mid SA \mid S \mid aB \mid a \\ A &\rightarrow S \mid B \\ B &\rightarrow b \end{aligned}$$

Let's remove the easy unit rule $S \rightarrow S$ first. It's redundant, I don't need to change anything else when removing this.

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow ASA \mid AS \mid SA \mid aB \mid a \\ A &\rightarrow S \mid B \\ B &\rightarrow b \end{aligned}$$

For the unit rule $S_0 \rightarrow S$, we can remove this by "duplicating" the productions of $S$ for $S_0$

$$\begin{aligned} S_0 &\rightarrow ASA \mid AS \mid SA \mid aB \mid a \\ S &\rightarrow ASA \mid AS \mid SA \mid aB \mid a \\ A &\rightarrow S \mid B \\ B &\rightarrow b \end{aligned}$$

I can apply this same process for the unit rules $A \rightarrow S$ and $A \rightarrow B$

$$\begin{aligned}
S_0 &\rightarrow ASA \mid AS \mid SA \mid aB \mid a \\
S &\rightarrow ASA \mid AS \mid SA \mid aB \mid a \\
A &\rightarrow ASA \mid AS \mid SA \mid aB \mid a \mid b \\
B &\rightarrow b
\end{aligned}$$

For the last step, we want to convert all the final rules to a proper form. Note how $S \rightarrow ASA$ does not follow the form $A \rightarrow BC$.

For each rule $X \rightarrow u_1 u_2 \dots u_k$ where $k \geq 3$ and each $u_i \in (V \cup \Sigma)$ we create the rules $X \rightarrow u_1 X_1$, $X_1 \rightarrow u_2 X_2$, $\dots$, $X_{k-2} \rightarrow u_{k-1} u_k$. If $k = 2$, we create the rule $U_i \rightarrow u_i$ and replace each occurance of $u_i$ to follow.

To apply this to $S \rightarrow ASA$, we add the rules $A \rightarrow AA_1$ and $A_1 \rightarrow SA$. I've also created a rule $U \rightarrow a$ to replace the rule $S \rightarrow aB$

Our final CFG in Chomsky Normal Form is:

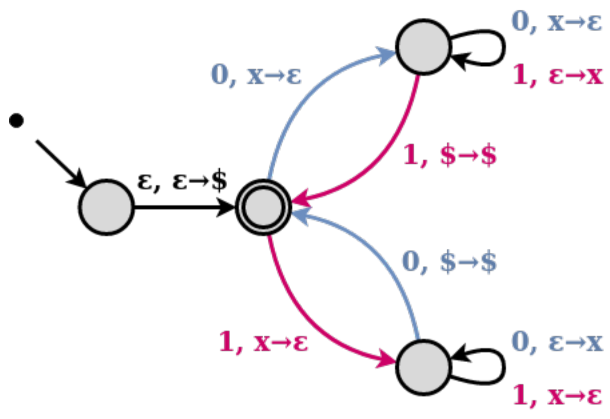
$$\begin{aligned}
S_0 &\rightarrow AA_1 \mid AS \mid SA \mid UB \mid a \\
S &\rightarrow AA_1 \mid AS \mid SA \mid UB \mid a \\
A &\rightarrow AA_1 \mid AS \mid SA \mid UB \mid a \mid b \\
A_1 &\rightarrow SA \\
U &\rightarrow a \\
B &\rightarrow b
\end{aligned}$$

## 6.2 Pushdown Automata

As DFA/NFAs are to regular languages Pushdown Automata (PDA) are to context free languages.

PDA can write to the stack, and read them back later. Just like the stack in any other CS context, you can push and pop symbols and its a last-in, first-out structure.

Remember that problem with the equal 0s and 1s and how thats non-regular, the following pushdown automata can recognize it:

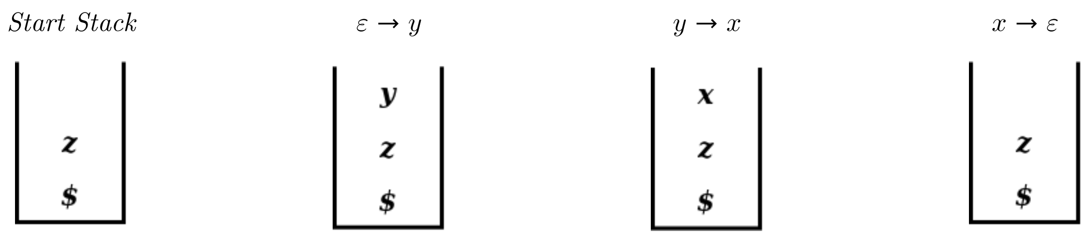


Transitions take a new format:

$$a, b \to c$$

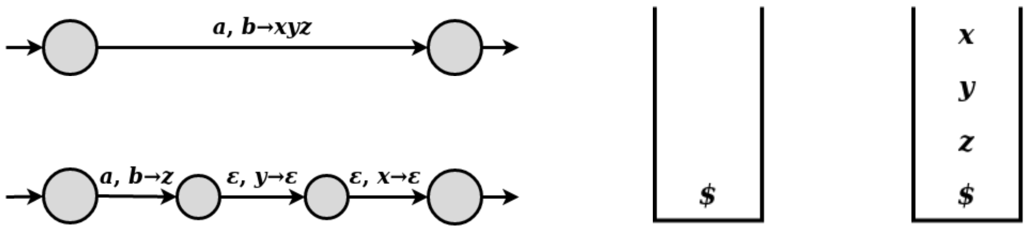
- $a$ is the letter read from input
- $b$ is what we are popping off the stack
- $c$ is what we are pushing onto the stack
- We can also say "$b$ replaces $c$" on the stack
- If $b$ doesn't exist on the stack, then we cannot transition

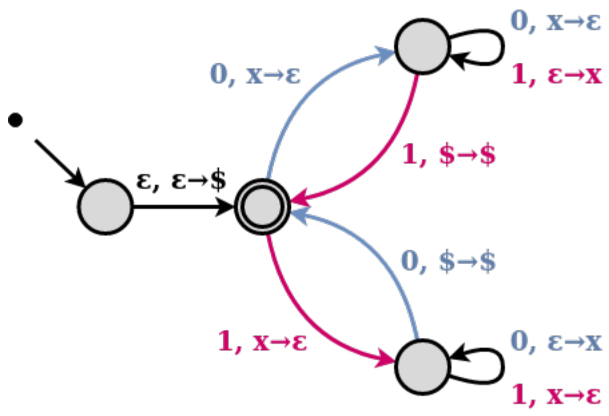
Reading a letter from input is the same as before. Here are a few example stack transistions:



Another thing to note, we need a flag to let us know when we're at the bottom of the stack. We can only check what the top character on the stack is, we can't check if the stack is empty or not. We use $ as our bottom flag. The first thing we need to do in a PDA is push a \$ on the stack. If we need to check if our stack is empty, we can check if the \$ is at the top.

We can also write a full string onto the stack in a single transition by using the shorthand $a, b \to xyz$. This is because we can expand this into many states:





Now to actually process the above PDA that recognizes the language of an equal number of 0s and 1s:

1. Depending on the first symbol from the string read, we assign one symbol as a "push symbol" and the other as a "pop symbol"
   - If $w \in 0\Sigma^*$ then 0 is the "push symbol" and 1 is the "pop symbol"
   - If $w \in 1\Sigma^*$ then 1 is the "push symbol" and 0 is the "pop symbol"
2. Everytime we encounter a "push symbol", we push an $x$ onto the stack
3. Everytime we encounter a "pop symbol", we pop an $x$ off the stack
4. We can only finish and accept the string when the stack is "empty" and we've seen an equal number of "push" and "pop" symbols (0s and 1s)

### 6.2.1 Formal Definition

Now a formal definition where a pushdown automata is a 6 tuple $A = (Q, \Sigma, \Gamma, \delta, q_0, F)$ where

- $Q$ is a finite set of states
- $\Sigma$ is a finite input alphabet
- $\Gamma$ is a finite stack alphabet
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to P(Q \times \Gamma_\varepsilon)$ is the transistion function
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is is the finite set of accept states

And the formal definition of computation is as follows where $w$ is the input string, $r$ is the sequence of states, and $s$ is the stack.

- $r_0 = q_0$ and $s_0 = \varepsilon$. This means M starts properly on the right start state with an empty stack
- For $i = 0, ..., m-1$ we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$ where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$ This is fancy math words for making sure M moves to the next state properly according to the state, stack, and next input symbol.
- $r_m \in F$ The accept state is the last state.

## 6.3 Converting CFGs to PDAs

Recall the formal definition of a CFG $G = (V, \Sigma, R, S)$ and the formal definition of a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$.

The easiest thing to start with when constructing $P$ is $q_0$ and $F$. There will be one start state ($q_0$) and one accept state ($q_a$). These will represent putting \$ on the stack and taking it off when we're done, only accepting when we have an empty stack.

$$q_0 = q_0$$

$$F = \{q_a\}$$

$Q$ will contain $q_0$ and $q_a$. It will also contain $q_l$ or $q_{loop}$. The process we follow when making a PDA from a CFG will be pushing and popping symbols in and out of one state until the stack is empty. As seen before with the string shorthand for stacks, we'll also need a set of "extra states" for the expanded shorthand which we'll call $E$. This makes:

$$Q = \{q_0, q_l, q_a\} \cup E$$

Another easy one, we know our input alphabet has the be the same alphabet as the CFG. Then based on the looping structure, the stack alphabet will be the alphabet of the CFG in addition to all the variables and the \$:

$$\Sigma = \Sigma$$

$$\Gamma = \Sigma \cup V \cup \$$$

The last thing to define is the transition function. Before I start, I'm gonna quick go over the transition function notation and the order of arguments (otherwise I will forget):

$$\delta(q_f, w_i, s_o) = \{(q_t, s_u)\}$$

$$\begin{aligned}
q_f &= \text{The "from" state} \\
w_i &= \text{The input letter being read} \\
s_o &= \text{The element on the stack being popped} \\
q_t &= \text{The "to" state} \\
s_u &= \text{The element being pushed on the stack}
\end{aligned}$$

Now, let's construct a transistion function. As stated before, the first thing we need to do is put \$ on the stack. We also need to put the starting variable $S$ on the stack too. This gives us a starting point to loop on.

$$\delta(q_0, \varepsilon, \varepsilon) = \{(q_l, S\$)\}$$

Let's also define the accepting transistion where we are popping the \$ flag and have an empty stack:

$$\delta(q_l, \varepsilon, \$) = \{(q_a, \varepsilon)\}$$

We then have two other "types" of transistions:

1. The top of the stack is a variable
2. The top of the stack is a terminal

In case 1, we have a variable on the top of the stack. This variable needs to be replaced with the things it produces. For each variable, we're going to create a transistion that consumes the variable from the stack and pushes its production onto the stack. This will create the "extra" shorthand states $E$ that will be unioned with $Q$. The general shorthand can be written as:

$$\delta(q_l, \varepsilon, A) = \{(q_l, w)\} \text{ for rule } A \rightarrow w$$

To formally expand this into the proper "extra" states and transistions, we'd have:

$$\begin{aligned}
w &= w_1 w_2 \dots w_k && \text{The input string of length } k \\
E &= E \cup \{e_1, e_2, \dots, e_{k-1}\} && \text{The extra states} \\
\delta(q_l, \varepsilon, A) &= \{(e_1, w_k)\} && \text{ for rule } A \rightarrow w \\
\delta(e_i, \varepsilon, \varepsilon) &= \{(e_{i+1}, w_{k-i})\} && \forall i \in [2, k-2] \\
\delta(e_{k-1}, \varepsilon, \varepsilon) &= \{(q_l, w_1)\}
\end{aligned}$$

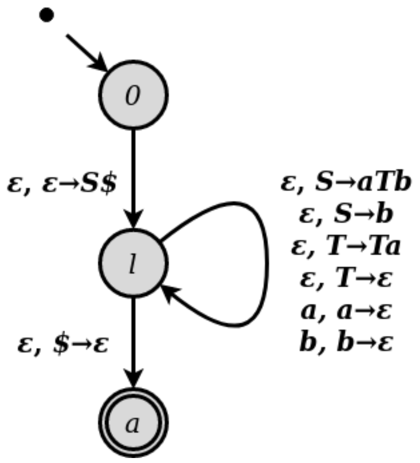
In case 2, where we have a terminal, its means that a variable had been seen earlier and told us to expect a terminal input. For the rule $S \rightarrow aTb$, we consume the $S$ and put $b$, $T$, and $a$ on the stack (in that order). Since the top of the stack is an $a$, we know we need to see an $a$ in input and consume it before processing $T$ or $b$. This gives the following transistion:

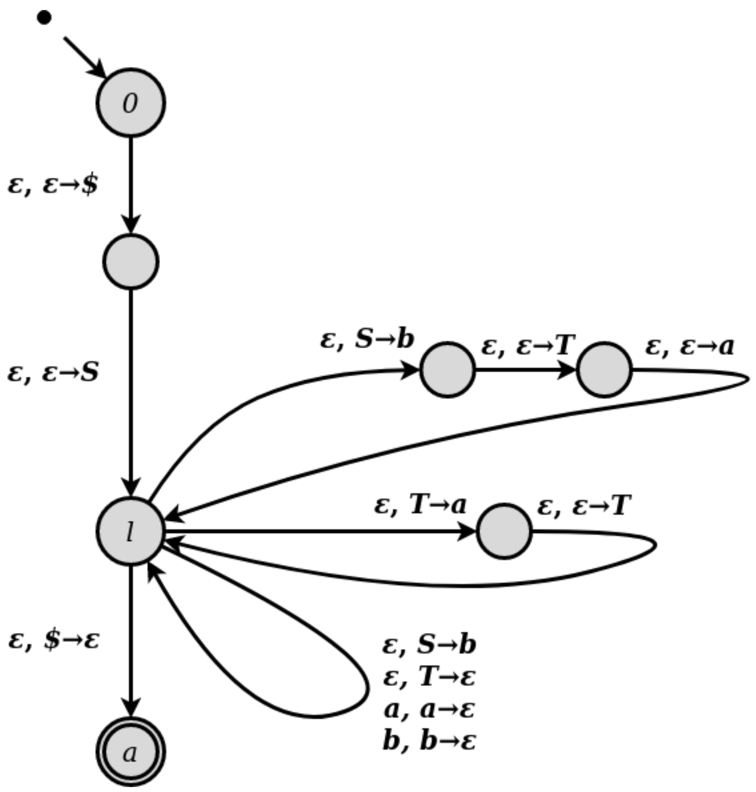
$$\delta(q_l, a, a) = \{(q_l, \varepsilon)\} \text{ for terminal } a$$

In practice, this looks like:

*Note: Pages 117-119 From Sipser*

$$S \to aTb \mid b$$
$$T \to Ta \mid \varepsilon$$





## 6.4 CYK Algorithm

The CYK algorithm takes a CFG $G$ in Chomsky Normal Form and checks if a string $w \in L(G)$.

String $w$ of length $n$ is $\in L(G)$ if and only if $S \in X_{1n}$. The algorithm constructing the set $X_{1n}$ out of $w$ operates $\in O(n^3)$. We construct each set by doing the following:

$$\begin{aligned} w &\neq \varepsilon \\ w &= a_1 a_2 \dots a_n \\ X_{ij} &= \{A \in V \mid A \overset{*}{\Rightarrow} a_i a_{i+1} \dots a_j\} \end{aligned}$$

We start with the base case where $X_{ii}$ the variable $A$ that creates the single character $a_i$

$$X_{ii} = \{A \in V \mid A \Rightarrow a_i\}$$

Inductively, we can then build $X_{ij}$:

$$X_{ij} = \bigcup_{k=i}^{j-1} \{A \in V \mid A \to BC, B \in X_{ik}, C \in X_{k+1j}\}$$

## 6.5 Pumping Lemma

Just as before with regular languages, there is a pumping lemma for context free languages.

If $A$ is a context free language, then there is a number $p$ (the pumping length) where if $s$ is any string in $A$ of at least length $p$, then $s$ can be divided into the following 5 part structure $s = uvxyz$ where

- for each $i \geq 0, uv^i xy^i z \in A$
- $|vy| > 0$
- $|vxy| \leq p$

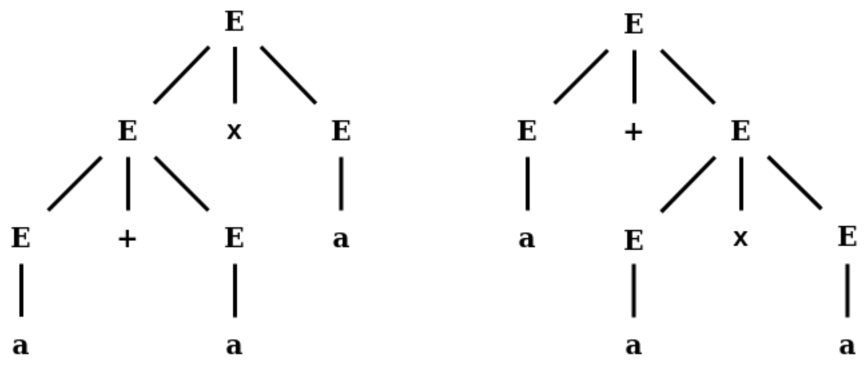
## 6.6 Ambigious Grammars

An ambigious Context Free Grammar is a CFG that can generate the same string in many different ways.

In the following grammar, $a + a \times a$ is generated ambiguously:

$$E \to E + E \mid E \times E \mid (E) \mid a$$

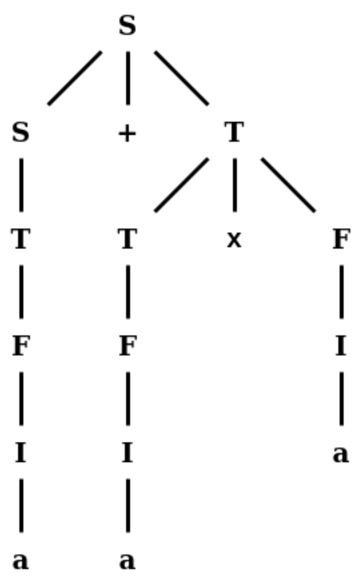
The following two parse trees are generated:



The unambigious version of this grammar is:

$$\begin{aligned}
S &\to T \mid S + T \\
T &\to F \mid T \times F \\
F &\to I \mid (S) \\
I &\to a
\end{aligned}$$

Here, we use $T$ to generate *terms* from addition, $F$ to generate *factors* from multiplication, and $I$ to generate *identifiers*. In this CFG, addition is the lowest priority. Using this CFG, only one parse tree can be generated:



A grammar is **inherently ambigious** if it cannot be written unambigiously. The following language cannot be written unambigiously:

$$L = \{a^i b^j c^k \mid i = j \lor j = k\}$$

| | |
|---|---|
| $S \rightarrow IC \mid AK$ | $i = j$ or $j = k$ |
| $I \rightarrow aIb \mid \varepsilon$ | $i = j$ |
| $C \rightarrow Cc \mid \varepsilon$ | A lot of $c$s |
| $K \rightarrow bKc \mid \varepsilon$ | $j = k$ |
| $A \rightarrow Aa \mid \varepsilon$ | A lot of $a$s |

In the case where $i = j = k$, the CFG can take the $i = j$ "path" through $IC$ or it can take the $j = k$ "path" through $AK$. Both can generate the $i = j = k$ string.

There is not an algorithm that can determine if a grammar is inherently ambigious, nor is there one to convert an ambigious grammar to an unambigious grammar.

## 6.7 Deteministic Context Free

We know that the difference between determinism and nondeterminism is the transistion function $\delta$. If $\delta$ goes to a single state, it's deterministic. If $\delta$ goes to a set of states, it's nondeterministic.

Deterministic regular languages and nondeterministic regular languages are equivalent.

Deterministic Turing recognizable languages and nondeterministic Turing recognizable languages are equivalent (covered later).

Deterministic context free languages and nondeterministic contex free languages are *not* equivalent.

Some properties of DCFLs:

- DCFLs are a proper subset of CFLs
- DCFLs are unambigious
- DCFLs are closed under complement
- DCFLs are not closed under union or intersection

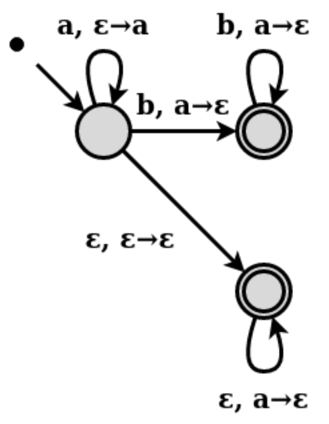
Looking at the language $L$, we can create a language $L\$$ where $w' \in L\$$ such that $w' = w\$$ where $w \in L$. A language $L$ is a DCFL if and only if $L\$$ is a DCFL.

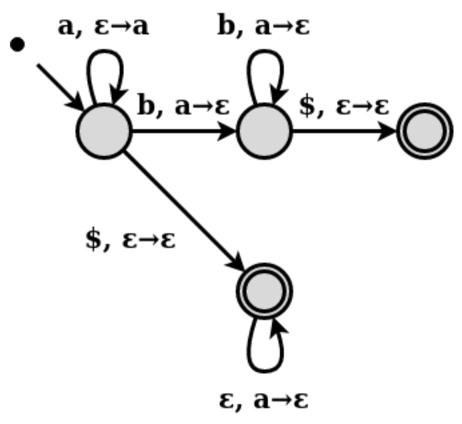
### 6.7.1 Deterministic Push Down Automata

A Deterministic Push Down Automata (DPDA) is a 6-tuple $A = (Q, \Sigma, \Gamma, \delta, q_0, F)$ such that:

- $Q$ is the set of states
- $\Sigma$ is the input alphabet
- $\Gamma$ is the stack alphabet
- $\delta$ is the transistion function defined as $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to P(Q \times \Gamma_\varepsilon)$ with an extra requirement that each assigned value contains at most one pair
  - $\forall q \in Q, a \in \Sigma$ and $x \in \Gamma$, exactly one of the following values is not the empty set $\varnothing$
    - $\delta(q, a, x)$
    - $\delta(q, a, \varepsilon)$
    - $\delta(q, \varepsilon, x)$
    - $\delta(q, \varepsilon, \varepsilon)$
- $q_0 \in Q$ is the start state
- $F \subset Q$ is the set of accepting states

Given the language $L = a^* \cup \{a^n b^n \mid n > 0\}$, below are two PDAs that accept the language. On the left, the NPDA and on the right, the DPDA.

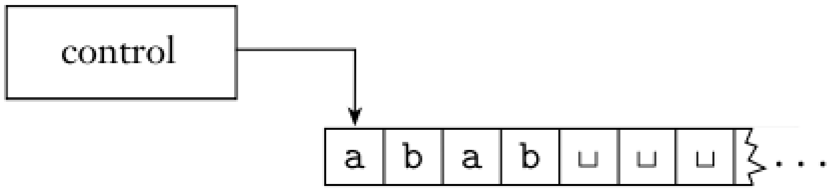




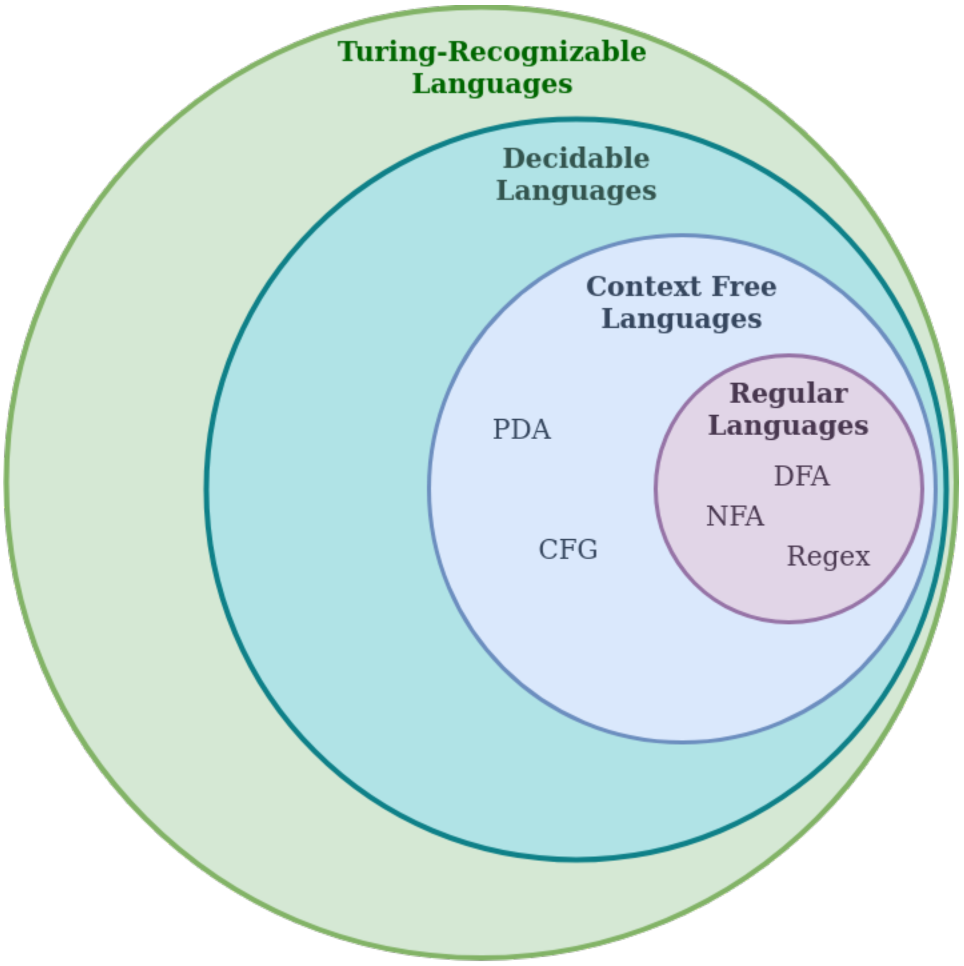
On the other hand, $L = \{a^i b^j c^k \mid i = j \wedge j = k\}$ cannot be deterministic context free because it is inherently ambigious

# 7 Turing Machines

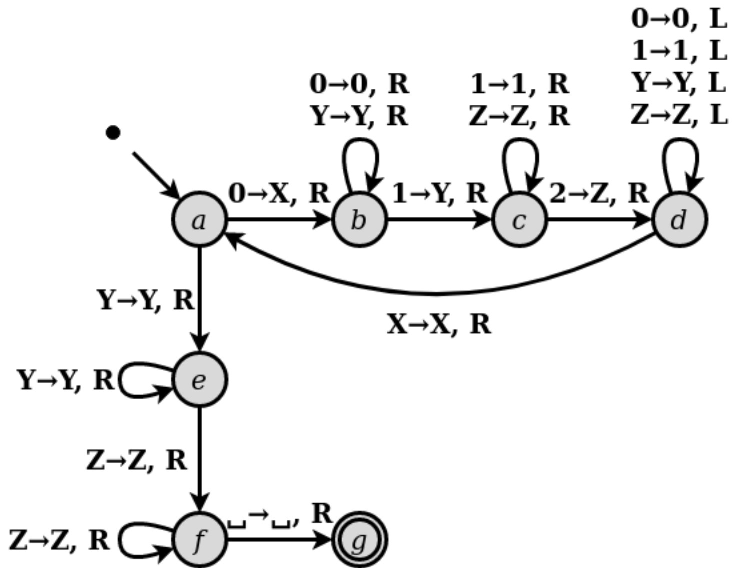
My homie Alan Turing thought of these bad boys. A Turing machine is a simple computer that uses an infinite tape as memory. This tape starts with the input string and is infinitely empty (␣) afterwards. The head can move left and right and read and write symbols. Reject and accept states happen immidiately.



We can now update our heirarchy we saw before with Turing Machines:



To show a TM recognizing a non context free language, let's walk through the computation of the language $L = \{0^n 1^n 2^n \mid n \geq 1\}$

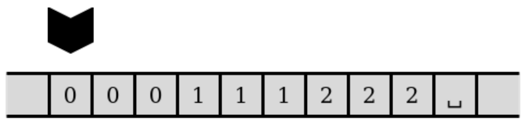


We'll notice transistions take a new format:

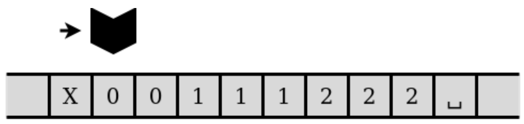
$$r \to w, d$$

- $r$ : The symbol on the tape underneath the head being read
- $w$ : The symbol being written on the tape which "overwrites" the existing symbol
- $d$ : The direction the head should move. The head will only move one space
  - $L$ : Left
  - $R$ : Right

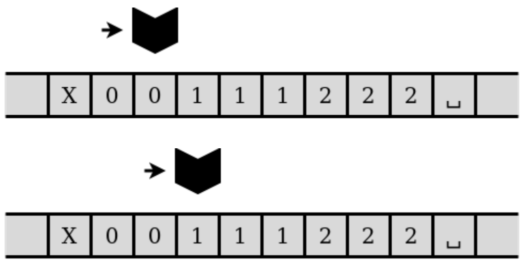
To read the string, we're going to load the full string into the tape followed by infinite blanks (␣).



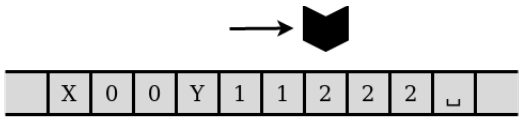
We're starting in state $a$. Underneath the head, there is a 0 which means we progress to state $b$. As we transistion, the 0 becomes an $X$ and the head moves right:



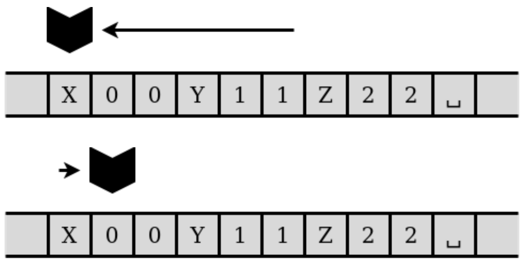
We're going to stick around in state $b$ beause we're going to read and "overwrite" these 0s with 0s. We're going to do this as long as we see a 0 or a $Y$. Essentially, we're just moving the head right until we read a 1:



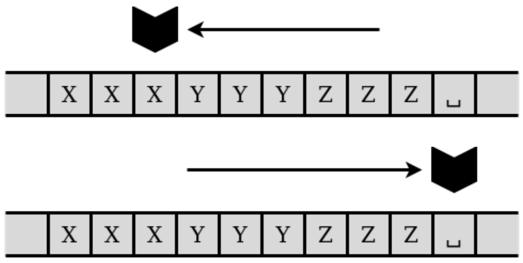
Now when we encounter a 1, we change it to a $Y$ and push on. The same shifting process will repeat. I'm going to start generalizing some of the more repetitive movements.



Now when we read a 2, we're going to move left until we read an $X$, then restart the process:



We do this until we run out of 0s. Once there are no more, we read $Y$s and $Z$s until we hit a ␣, which is when we finally accept the string!



As seen here, I was able to generalize some of the transitions by describing the behavior of the TM. In this class, we're not going to have to write out formal definitions of Turing Machines with states and functions. We'll be able to use algorithmic steps to describe them instead.

## 7.1 Deterministic Turing Machines

The formal definition of a Detministic Turing Machine is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ such that:

$Q$ The set of states
$\Sigma \subset \Gamma$ The input alphabet
$\Gamma$ The tape alphabet. It also includes the blank symbol
$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
$q_0 \in Q$ the start state
$q_a \in Q$ the accept state
$q_r \in Q$ the reject state

## 7.2 Computing

**Turing-Recognizable :** A language is Turing-Recognizable if some Turing machine recognizes it

**Decider :** A Turing machine that halts on all inputs

**Turing-Decidable :** A language is Turing-Decidable (or just decidable) if there is a decider $M$ recognizing it

**Configuration :**

Configurations of $M$ have the form $uqv$, where $u, v \in \Gamma^*$ and $q \in Q$.

There are four types of configurations:

1. Start
2. Accepting
3. Rejecting
4. Halting

A Turing Machine has three possible outputs. It can accept input, reject input, or not halt and loop infinitely.

A Turing Machine $M$ accepts string $w$ such that there exists a sequence of configurations $C_1, C_2, \ldots, C_k$ such that:

1. $C_1$ is the start configuration of $M$ on $w$
2. Each $C_i$ yields $C_{i+1}$
3. $C_k$ is an accept configuration

## 7.3 Multitape Turing Machines

We can create a Turing Machine with multiple tapes. We redefine $\delta$ as the following:

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k$$

Which looks like:

$$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, A_1, \ldots, A_k)$$

Where:

$$\begin{aligned} q_i &\in Q \\ q_j &\in Q \\ a_m &\text{ a tape symbol} \\ b_m &\text{ a tape symbol} \\ A_m &\text{ a direction} \end{aligned}$$

### 7.3.1 Converting a Multitape TM to a Single Tape TM

## 7.4 Enumerators

An enumerator is basically a Turing Machine with a printer attached. The printer will print and output strings.

A language is Turing-recognizable if and only if some enumerator enumerates it. ‘