CHAPTER 1: INTRO TO THE THEORY OF COMPUTATION

» Introduction

- The theoretical foundation of computer science is made up of the following:
  - Automata theory
  - Formal languages
  - Grammars
  - Computability
  - Complexity
- ○ **Automaton**
  - Is a construct that possess all the indispensable features of a digital computer
  - Accepts input, produces output and can have storage and make decisions
- ○ **Formal language**
  - An abstraction of the general characteristics of programming languages
  - Consists of a set of symbols can be combined into entities called 'sentences'
  - A formal language is the set of all sentences permitted by those rules

» 1.1 Mathematical preliminaries and notations

- Collection of elements
- To indicate x is an element of S, we write $x \varepsilon S$
- To indicate x is not an element of S, we write $x \not\varepsilon S$
- ○ **Sets**
  - $S = \{0,1,2\}$, $S = \{i : i > 0, i \text{ is even}\}$ //examples of sets
  - The second notation will read like this
    - "S is the set of all I, such that i is greater than 0 and i is even"
    - $S_1 \cup S_2 = \{x : x \varepsilon S_1 \text{ or } x \varepsilon S_2\}$
    - $S_1 \cap S_2 = \{x : x \varepsilon S_1 \text{ and } x \varepsilon S_2\}$
    - $S_1 – S_2 = \{x : x \varepsilon S_1, \text{ but not in } S_2\}$ //take the set of S_1 and take away all elements of S_2
    - The complementation of a set is everything that isn't in it
      - › $!S = \{x : x \varepsilon \text{ Universe, and not in } S)$
    - $S \cup 0 = S$
    - $S \cap 0 = 0$
    - $!!S = S$
    - 0 is the empty set, the complement of 0 is the universe of all elements

» 1.1 Mathematical preliminaries and notations

- **sets**
  - S_1 is a subset of S_2 then S_1 ε S_2
  - Disjointed sets are set without common elements and therefore $S_1 \cap S_2 = 0$
  - A set is finite if it contains a finite number of elements
  - The size of a set is denoted by |S|
  - Given a set 'S', it has many subsets, the set of all subsets is called the powerset and denoted by this : 2^S

If S is the set $\{a, b, c\}$, then its powerset is

$$2^S = \{\varnothing, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}.$$

Here $|S| = 3$ and $|2^s| = 8$. This is an instance of a general result; if S is finite, then

$$|2^S| = 2^{|S|}.$$

  - Cartesian product is the multiplication of two sets:

Let $S_1 = \{2, 4\}$ and $S_2 = \{2, 3, 5, 6\}$. Then

$$S_1 \times S_2 = \{(2, 2), (2, 3), (2, 5), (2, 6), (4, 2), (4, 3), (4, 5), (4, 6)\}.$$

- **Functions and relations**
  - Function is a rule that assigns to elements of one set a unique element to another set $f: S_1 \rightarrow S_2$

» 1.2 three basic concepts

- (1) Languages (2) Grammars (3) Automata
- **Languages**
  - The alphabet is a finite, non-empty set ∑ of symbols
  - From the individual symbols we construct strings
    - Strings being a finite sequence of symbols from eh alphabet
    - ∑ = {a,b} then ab, aba, aaa, bb, ababa, ababa, are all string of ∑
  - w = abaaa v = baba
  - The concatenation of w and v; wv = abaaababa //think concatenating string in java
  - The reverse of a string denote by w^R is: aaaba
  - Length of a string is |w| = 5
  - The empty string is λ |λ| = 0,
  - If w = uv, then u and v are substrings of w, u being the prefix and v being the suffic
  - |uv| = |u| + |v|

» 1.2 three basic concepts

- **Languages**
  - If we is a string w^n is the string we repeated n times, $W^0 = \lambda$, $w^1 = w$,
  - If ∑ is an alphabet, then we use ∑* to denote the set of string obtained by concatenating zero or more symbols
  - $\sum \{a,b\} \rightarrow \sum^* = \{\lambda, a, b, aa, bb, ab, ba, aaa, abb, aab, baa, bba, bbb, \ldots\}$
  - The set {a,aa,aab} is a language on ∑, since it has a finite # of sentences using the same ∑
  - $L = \{a^n b^n : n \geq 0\}$ is a language that is not finite (infinite language)
  - $!L = \sum^* - L$
  - $L^R = \{w^R : w \varepsilon L\}$ //reversing a language is just reversing every string in that language .
  - The concatenation of two languages L_1 and L_2 is the set of all string obtained by concatenating any element of L_1 with any element of L_2
    - Creating a new language with any possible combination of L_1 followed by L_2
    - $L_1 = \{a, bba\}$ $L_2 = \{b, aa, bb\}$
    - $L_1L_2 = \{ab, aaa, abb, bbab, bbaaa, bbbba\}$
    - $L_1^2 = \{aa, abba, bbaa, bbabba\}$
- **Grammar**
  - Grammar is used in the English language to create well-formed sentences
  - In computer science grammar does the same thing but on a different language

Definition 1.1

A grammar G is defined as a quadruple

$$G = (V, T, S, P),$$

where V is a finite set of objects called **variables**,
T is a finite set of objects called **terminal symbols**,
$S \in V$ is a special symbol called the **start** variable,
P is a finite set of **productions**.

It will be assumed without further mention that the sets V and T are nonempty and disjoint.

Definition 1.2

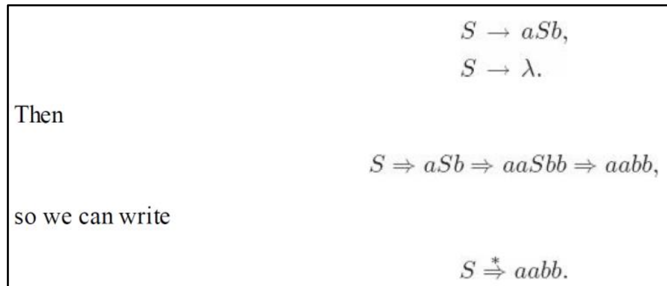
Let $G = (V, T, S, P)$ be a grammar. Then the set

$$L(G) = \left\{ w \in T^* : S \overset{*}{\Rightarrow} w \right\}$$

is the language generated by G.

  - Grammars are equivalent if $L(G_1) = L(G_2)$
    - Two separate grammars that make up the same language

- **<u>1.2 three basic concepts</u>**
  - **grammars**
    - As the two definitions explain grammars have variables, which you will see are associated with a pathway and then terminal symbols which related to elements in the alphabet



$$S \rightarrow aSb,$$
$$S \rightarrow \lambda.$$

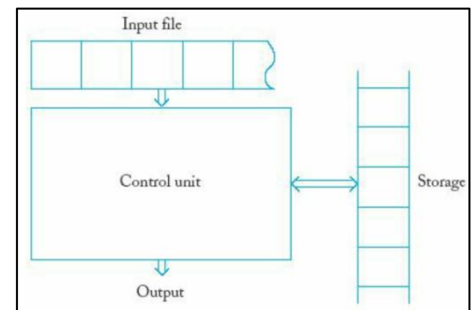
Then

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb,$$

so we can write

$$S \stackrel{*}{\Rightarrow} aabb.$$

  - **Automata**
    - An abstract model of a digital computer
    - Finite state machine will be the main focus
    - DFA = every possibility is explicitly shown and there are only one possibility for each element
    - NFA = only the successful paths are shown and multiple paths can extend from each node for an element
      - In an NFA if you reach a node and there is no path for an element, that means that "string" will fail on the language



Chapter 2: Finite Automata

» 2.1 Deterministic finite Accepters (DFA)

- **Deterministic accepters and transition graphs**
  - A DFA functions by traversing through a graph
  - If a string ends on a final node, it is accepted, it not, it is rejected
  - Δ (q0, a) = q1 : this means if you are at q0 and the next element in the string is an a, move to q1
  - Example of a DFA is shown on the right

Definition 2.1

A **deterministic finite accepter** or **dfa** is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where

Q is a finite set of **internal** states,

Σ is a finite set of symbols called the **input alphabet**,

$\delta : Q \times \Sigma \to Q$ is a total function called the **transition function**,

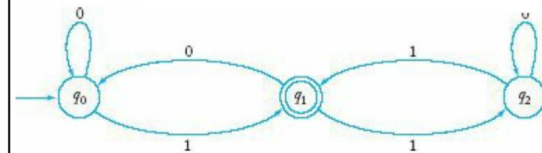
$q_0 \in Q$ is the **initial state**,

$F \subseteq Q$ is a set of **final states.**

$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$

where δ is given by

$\delta(q_0, 0) = q_0, \quad \delta(q_0, 1) = q_1,$
$\delta(q_1, 0) = q_0, \quad \delta(q_1, 1) = q_2,$
$\delta(q_2, 0) = q_2, \quad \delta(q_2, 1) = q_1.$



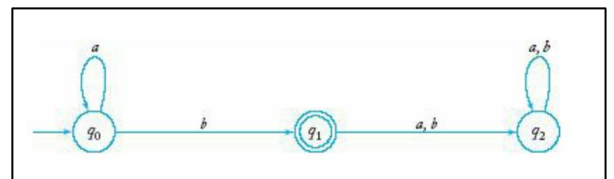
- **Languages and DFAs**

Definition 2.2

The language accepted by a dfa $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M. In formal notation,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}.$$

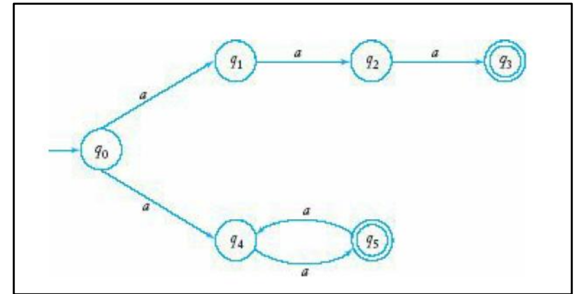
  - Trap state: a state in which if the input leads to it, it will never leave
  - in the DFA on the right q2 is a trap state
    - This is based on a ∑ {a,b}



- **Regular languages**
  - A language L is called regular if and only if there exists some deterministic finite accepter M such that L = L(M)
  - If L is regular so is L^2 and L^3…. L^n

» 2.2 Nondeterministic Finite Accepters (NFA)

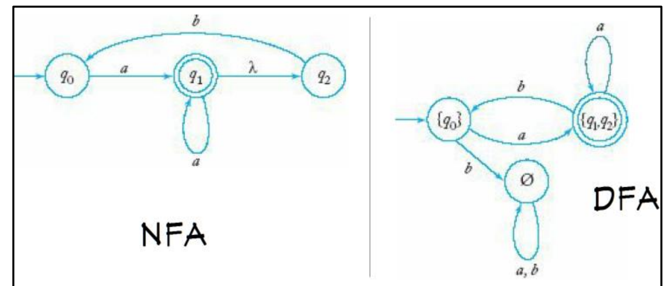
- A choice of moves for an automaton, rather then having a unique path for each element
- All NFA's can be changed into DFAs and all NFA are therefore regular languages
- in the example on the right we can see that on q0, if we have an a, we can either move to q1 or q4. This choice makes it non deterministic



- There other thing is the λ transition can be used
- Also only the successful entries must be shown
- No need to show that unsuccessful attempts go into a trap state
  - Why nondeterminism?
    - Because you could have multiple options at any state, therefore you try each state to see if one is successful
    - If one is successful then the string is successful!
    - Nondeterminism is therefore much easier to design even though it has the same power as a DFA

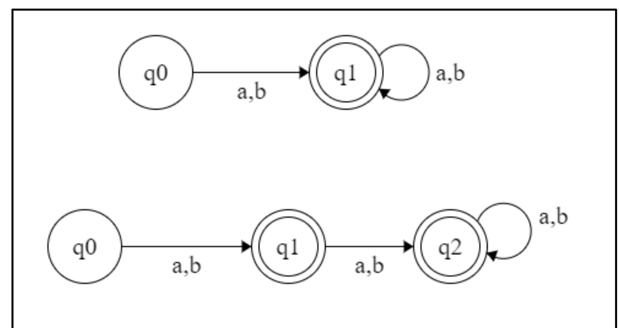
» 2.3 Equivalence of DFAs and NFAs

- Two FA's are said to be equivalent if they accept the same language
- There are multiple DFAs and NFAs for a single language
- Every DFA can be turned into an NFA and vice versa
- Example on the right →
    - Both accept the language
        - L = a(a + ba)*

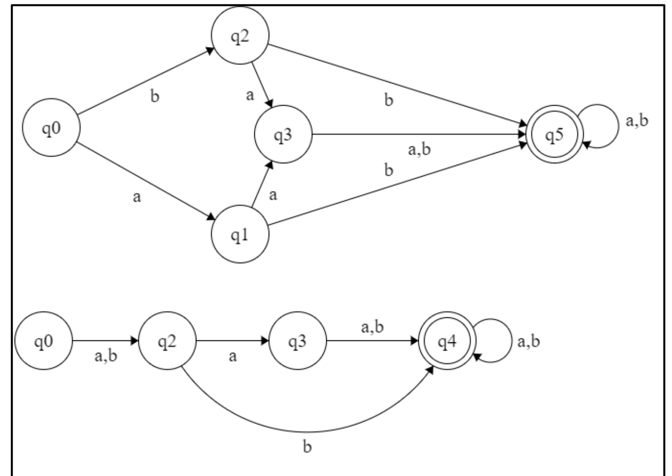


» 2.4 Reduction of the number of state in a FA

- Every DFA has a unique languages, but a languages does not have a unique DFA
    - Multiple DFA's for a language
- As you can see, the second DFA can be reduced to the first one as the states are redundant
- q1 to q2, is a redundant transition



- **how to do it?**
  - Remove all inaccessible states
  - Find state that have the same destination paths for all elements and merge the incoming paths,
  - As you can see, q1 and q2 have the same paths for a and b, therefore merge q1 and q2
  - L($\hat{M}$) is the machine the minimalist amount of states that will accept the language



» 3.1 Regular Expressions

- **Formal definition of a regular expression**
  - Let ∑ be a given alphabet, then,
    - 0, λ and a ε ∑ are all reg expressions
    - If r1 and r2 are regex, so are r1 + r2, r1r2, r1* and (r1)
    - A string is a regex if an only if it can be derived from a primitive regex by a finite number of applications
    - Example:
      - › (a + b + c)*(c + 0)   r1 = (a + b + c)*   r2 = (c + 0)
      - › L = r1*r2
- **Languages associated with regex**

Definition 3.2

The language $L(r)$ denoted by any regular expression r is defined by the following rules.

1. Ø is a regular expression denoting the empty set,
2. λ is a regular expression denoting $\{\lambda\}$.
3. For every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.

If r_1 and $r_2$are regular expressions, then

4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 \cdot r_2) = L(r_1) \cup L(r_2)$;
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

The last four rules of this definition are used to reduce $L(r)$ to simpler components recursively; the first three are the termination conditions for this recursion. To see what language a given expression denotes, we apply these rules repeatedly.

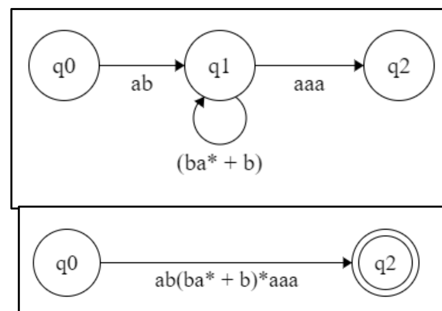
  - Concatenation precedes union
    - aa ∪ b = (aa) ∪ b   → {aa.b}   //NOT a (a ∪ b)
- **regex associated with languages**
  - L = {w ε {0,1} : w has at least 1 pair of consecutive 0}
    - L(r) = (0 + 1)* 00 (0 + 1)*
      - › the * means 0 or more iterations of that combination

» 3.1 Regular Expressions

- **regex associated with languages**
  - L = {w ε {0,1} : w has no consecutive 0}
    - L(r) = 1*(011*)*(λ + 0)
      - › start with 0 or more 1's
      - › followed by 0 or more combination of 01^n where n > 0
      - › followed by a zero, or lambda

» 3.2 Regex and Languages

- **Languages**
  - For every regular language there is a regex and vice versa
- **Regex denote regular languages**
  - Since a regular language accepts a DFA and every DFA has a equivalent NFA, we then can prove a regex is a regular by creating an NFA



- **regex for regular languages**
  - how to convert from a reg language to regex
    - using a generalized transition graph (GTG)
      - › which is basically an NFA with regex
      - › example of this is on the right
    - after that you expand each expression by removing nodes until you only have the initial state and the final state

» 3.3 Regular grammars

- So far to prove a language is regular we either create an NFA, DFA or write a regular expression
- The 4th way is to create a Right or Left Linear grammar (RLG or LLG)
- A grammar is the production possibility from each node written like this
  - S → Aaa
  - A → b|a
      - › This basically capital letters are variables and lower case letters are elements
      - › Replace variables with their production until you reach an answer
      - › Aaa → (b|a)aa → baa or aaa.

» 3.3 Regular grammars

- **Right and left linear grammars**

Definition 3.3

A grammar $G = (V, T, S, P)$ is said to be **right-linear** if all productions are of the form

$$A \to xB,$$

$$A \to x,$$

where $A, B \in V$, and $x \in T^*$. A grammar is said to be **left-linear** if all productions are of the form

$$A \to Bx,$$

or

$$A \to x.$$

A regular grammar is one that is either right-linear or left-linear.

- A language is regular if it has either a left or right linear grammar
- A left linear grammar is one that has all variables on the left hand side of each production
  - S → Aa
  - A → Baa | b
  - B → a |Abb
- A right linear grammar is one that has all variables on the right hand side of each production
  - S → aA
  - A → aaB | b
  - B → a |bbA
- Right and left linear grammars are called regular grammars → regular languages
- a grammar that isn't linear can be one that has either production that have
  - two variables in it, //S → aBB
  - a variable in between two elements //S → aBa
  - having both right and left productions // S→ aB, B → Aa

- Right linear grammars generate regular languages
  - To prove that a RLG is regular we can create and NFA for it
- Equivalence of regular languages
  - Right vs left linear grammars, we can say any languages has either a left or right linear grammar
  - Each left grammar can be changed to a right linear grammar

» 3.3 Regular grammars

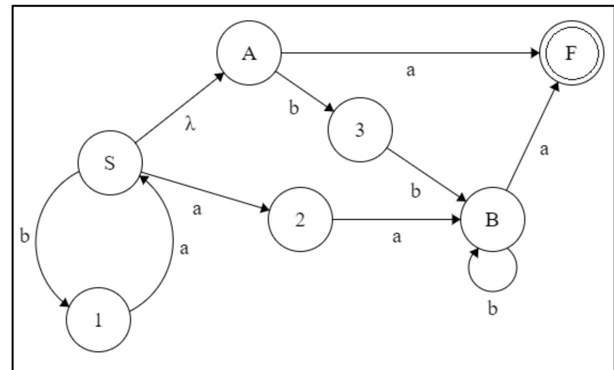
- Equivalence of regular languages
  - Very difficult to create an NFA of a left linear grammar
  - We convert a LLG to a RRG and then create an NFA that we reverse
    - How to convert from LLG to NFA
      - Step 1: reverse the productions
        - › S → Sab | Baa | A
        - › A → a | Bbb
        - › B → Bb | a

S → baS | aaB | A

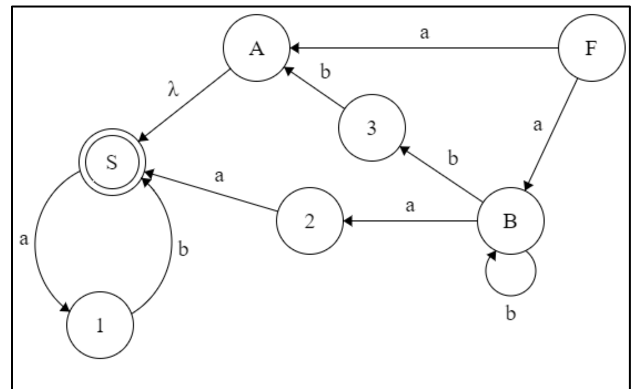
S → a | bbB

S → bB | a

      - Step 2: create NFA



      - Step 3: switch final state with
        the initial state and the direction of all productions



Chapter 4: Properties of Regular Languages

» 4.1 Closure Properties of regular languages

- Given L_1 and L_2 are regular, is $L_1 \cup L_2$ regular?
  - Obviously, if they are both regular than one or the other will be regular
  - This chapter will focus on all the closure properties of regular languages
- ○ **Closure under simple set operations**
  - If L_1 and L_2 are regular than the following expression are also regular

1. $L_1 \cup L_2$, //if both are regular, than clearly one or the other is regular
2. L_1L_2 //regular as you are adding one regular to another regular
3. L_1* //regular since it is just adding a regular language to the end of it * times
4. $!L_1$ //creating a DFA and then switching all regular states to final states and vice versa
5. $L_1 \cap L_2$, //DeMorgan's law: $L_1 \cap L_2 = !(L_1 \cup L_2)$ which is regular because of rule 1 and 5
6. $L_1 – L_2$ //this is the same as $L_1 \cap !L_2$, and because of rule 6 and 5 it is regular
7. L_1^R //creating an NFA and reversing it
8. L_1/L_2 //since L_3L_2 is regular if they are both regular, then think of L_1 being L_3L_2
   // clearly $L_1 / L_2 = L_3$

- ○ **Closure under other operations**
  - Homomorphism: when an alphabet is a reference to another alphabet
    - $\sum_1 = \{a,b,c\}$ $\sum_2 = \{a,b\}$
    - each letter in $\sum_1$ reference either a letter or string in $\sum_2$
    - ex… h(a) = abbbb, h(b) = λ, h(c) = bbaa
    - then if the word w is abc, the homomorphism is h(w) = abbbbbbaa
    - w = cbbbbb → h(w) = bbaa

» 4.2 Elementary Question about regular languages

- ○ **Theorem 4.6**
  - testing regular languages: use NFAs
  - there exists an algorithm for determining whether a regular language is empty, finite or infinite
  - if there is a path to a final vertex, or initial vertex is a final vertex, than it is not empty ($0 \neq \lambda$)
    - empty set = 0 set with and empty string = λ
  - if there is a loop between the initial state and the final state, than it is infinite
- ○ **Theorem 4.7**
  - There is an algorithm for determing if $L_1 = L_2$
    - $L_3 = (L_1 \cap !L_2) \cup (!L_1 \cap L_2)$ //L_3 would be empty if $L_1 = L_2$

» 4.3 identifying non-regular languages

- Regular languages can be infinite
- All finite languages are regular
  - **Using the pigeonhole principle**
    - Refers to the simple observation:
      - If we put n objects into m boxes, and if $n > m$, then at least one box must have more than one item in it
      - $L = \{a^n b^n : n \geq 0\}$ // how do you prove this isn't regular?
      - Since the number of machine states are finite and a^n is infinite there must be a state in M where the pigeonhole principle tells us that there must be some state say q that:
        - › $\delta(q0, a^n) = q$, $\delta(q0, a^k) = q$, //where $a^n \neq a^k$
        - › But since M accepts $a^n b^n$ we must have the transition : $\delta(q, b^n) =$
  - **A pumping lemma**
    - Uses the pigeonhole principle in another form
    - Let L be an infinite regular language, then there exist some positive integer m such that any w ε L $|w| \geq m$ can be decomposed as, $w = xyz$ with $|xy| \leq m$, and $|y| \geq 1$, such that $w_i = xy^i z$, is also in L for all i = 0, 1, 2, …
    - Every sufficiently long string in L can be broken into three parts in such a way that an arbitrary number of repetitions of the middle part yields another string in L
    - We say that the middle string is "pumped" hence the term pumping lemma for this result
    - Proof:
      - If L is regular, there exists a dfa that recognizes it. Let such DFA have states labeled q0, q1, q2…. Qn. Now take a string w in L such that $|w| \geq n + 1$
      - Since L is assumed to be infinite, this can always be done. No matter how big n is, the length of W can always be greater
    - The demonstration is always proof by contradiction
      - □ Example: $L = \{a^n b^n : n \geq 0\}$
        - Step one assume L is regular
        - We don't know the value of m, but whatever it is, we can always choose n = m
        - Therefore the substring y must consists of entirely of a's $|y| = k$
        - $W = xyz$, $|xy| \leq m$, $|y| \geq 1$, $w = xy^i z$ with $xy = a^{m-k}a^k$, $z = b^n$
        - $W = a^{m-}k(a^k)^i b^m$ now If i = 0, we have $a^{m-k}b^m$ and clearly $m-k \neq m$ if $k \geq 1$

- **4.3 identifying non-regular languages**
  - **A pumping lemma**
    - In applying the pumping lemma, we must keep in mind what the theorem says
      - We are guaranteed the existence of an m as well as the decomposition of xyz, but we do not know what they are
      - We cannot claim that we have reached a contradiction just because the pumping lemma is violated for some specific values of m or xyz
      - If the pumping is violated even for one w or i, then the language cannot be regular
    - Think of it this way: imagine playing a game
      - The opponent chooses a value for m
      - Given m, we pick a string in w in L of length equal or greater than m, we are free to choose any w, subject to w ε L and $|w| \geq m$
      - The opponent choose the decomposition xyz, subject to $|xy| \leq m$, $|y| \geq 1$
        - They will choose an option that makes the most sense for them
      - We try to pick i in such a way that the pumped string w_i is not in L
    - $L = \{n_a(w) < n_b(w)\}$ show that this isn't regular
      - Create a w in L that will give us an advantage → $w = a^m b^{m+1}$
        - Why does this give us the advantage?
          - Since $|xy| \leq m$, that means that |xy| can only be a's
          - Therefore, $w_i = xyz \rightarrow a^m b^{m+1} \rightarrow a^{m-k}(a^k)^i b^{m+1}$
        - Now we have to choose a y that makes that the # of a's equaled to the # of b's
        - If we choose i = 2 we get → $a^{m+k}b^{m+1}$, and we know that $k \geq 1$, therefore m + K is not < m +1 → contradiction → not regular!

» 5.1 Context Free Grammars

- A regular grammar is restricted in two ways
  - The left side must be a single variable
  - The right side must be either right linear or left linear
- To create more powerful grammars (grammars that are not regular), we must remove some of these rules
  - By removing the right and left linear rule, and only adhering to the rule of having one variable on the left hand side of the grammar, we have now created a Context Free Grammar (CFG) which is a context free language (CFL)

Definition 5.1

A grammar $G = (V, T, S, P)$ is said to be **context-free** if all productions in P have the form

$$A \to x,$$

where $A \in V$ and $x \in (V \cup T)^*$.

A language L is said to be context-free if and only if there is a context-free grammar G such that $L = L(G)$.

- It's simple to understand that every regular grammar is context free
- But being that it also accepts non left or right linear languages, grammars that are no regular, which mean languages that are not regular can also be accept.
  - □ Ex. $L = \{a^n b^n : n \geq 0\}$ this can be shown by the following grammar
    - $S \to aSb \mid \lambda$
  - □ Ex. $L = \{ww^R : w \in \{a,B\}^*\}$
    - $S \to aSa \mid bSb \mid \lambda$
  - ○ **Context free languages can be linear or not linear**
    - A non-linear grammar uses two variables in the same function
      - □ $S \to aSb \mid SS \mid \lambda$
  - ○ **Leftmost and rightmost derivations**

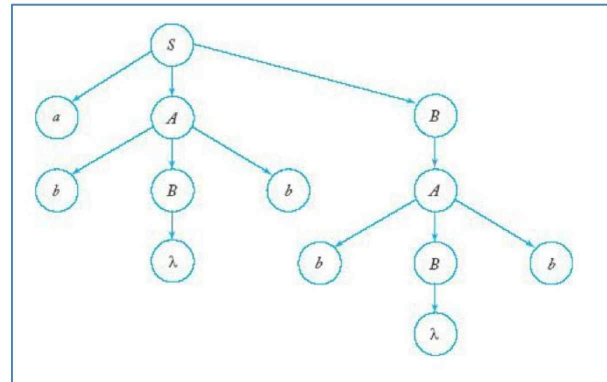
Definition 5.2

A derivation is said to be **leftmost** if in each step the leftmost variable in the sentential form is replaced. If in each step the rightmost variable is replaced, we call the derivation **rightmost**.

- Always replacing the left most or right most variable, and then replacing the final variable at the end

- Different ways to operate in working out not linear grammars

- **Derivation Trees**
  - You can also showing non-linear grammars is by using trees
  - Breaking the tree down into parts and then added them all up to form the total string
    - Ex.
      - $S \rightarrow aAB$,
      - $A \rightarrow bBb$,
      - $B \rightarrow A|\lambda$.



- **Relation between sentential forms and derivation trees**
  - Derivation trees are a very explicit and easy way to comprehend a grammar
  - For every grammar there exists a tree

» <u>**5.2 Parsing and ambiguity**</u>

- Whether or not w is in L(G)
  - **Parsing and membership**
    - Systematically constructing all possible derivations of L(G) to see if any match the word *w* that is in question
    - Ex. Consider the grammar
      - $S \rightarrow SS \mid aSb \mid bSa \mid \lambda$. And the grammar w = aabb
    - This give us the following possibilities by replacing the left most variable
      - $S \rightarrow SS \rightarrow SSS$,
      - $S \rightarrow SS \rightarrow aSbSS$,
      - $S \rightarrow SS \rightarrow bSaSS$,
      - $S \rightarrow SS \rightarrow S$,
    - Repeat the process with aSb and bSa
      - $S \rightarrow aSb \rightarrow aSSb$,
      - $S \rightarrow aSb \rightarrow aaSbb$,
      - $S \rightarrow aSb \rightarrow abSab$,
      - $S \rightarrow aSb \rightarrow ab$,
      - $S \rightarrow bSa \rightarrow bSSa$,
      - $S \rightarrow bSa \rightarrow baSba$,
      - $S \rightarrow bSa \rightarrow bbSaa$,
      - $S \rightarrow bSa \rightarrow ba$.
    - As you can clearly see, aabb is one of the strings made by this grammar.
    - While search parsing will get you the right answer, it is very slow and inefficient and can get stuck in a loop searching for the wrong answer

Definition 5.4

A context-free grammar $G = (V, T, S, P)$ is said to be a **simple grammar** or **s-grammar** if all its productions are of the form

$$A \to ax,$$

where $A \in V$, $a \in T$, $x \in V^*$, and any pair (A, a) occurs at most once in P.

- A simple grammar can only have one iteration of an element per variable on the left hand side
  - $S \to aS \mid bSS \mid \lambda$ //this is a simple grammar
  - $S \to aS \mid bSS \mid aS \mid \lambda$ //this is not a simple grammar

○ **Ambiguity in Grammars and Languages**

Definition 5.5

A context-free grammar G is said to be **ambiguous** if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

- If you can get to the answer two different ways, than it is ambiguous
- In regular languages this ambiguity is tolerated, however in programming languages this type of ambiguity must be completely removed

Definition 5.6

If L is a context-free language for which there exists an unambiguous grammar, then L is said to be unambiguous. If every grammar that generates L is ambiguous, then the language is called **inherently ambiguous**.

It is a somewhat difficult matter even to exhibit an inherently ambiguous language. The best we can do here is give an example with some reasonably plausible claim that it is inherently ambiguous.

» 5.3 Context-Free Grammars and Programming Languages

- the importance of the theory of formal languages is the definition of programming languages and in the construction of interpreter and compilers for them

» 6.1 Methods for Transforming Grammars

- While a CFG has no restriction to the right hand side of the equation, it does have some guidelines to follow to build more efficient grammars
- There are also 2 types of normal CFG
  - Chomsky normal form
  - Greibach normal form
- The empty string plays a singular role and therefore we try eliminating it by adding a pre node
  - $S_0 \rightarrow S \mid \lambda$ //this now adds a node that accepts λ but also splits off to the actual path
  - **A useful substitution rule**
    - A production $A \rightarrow x_1Bx_2$ can be eliminated from a grammar if we put in its place the set of productions in which B is replaced by all strings it derives in one stop
      - Ex.
        - A → a | aaA | abBc,
        - B → abbA | b.
          - › This can be changed to the following:
        - A → a | aaA | ababbAc | abbc.
  - **Removing useless productions**
    - Removing productions that can never take place in the derivation, either because they are unreachable or infinite loops
      - S → aSb | λ | A,
      - A → aA, //this is an infinite loop and therefore should never be reached
    - Update the new Grammar to this
      - S → aSb | λ

Definition 6.1

Let $G = (V, T, S, P)$ be a context-free grammar. A variable $A \in V$ is said to be **useful** if and only if there is at least one $\omega \in L(G)$ such that

$$S \overset{*}{\Rightarrow} xAy \overset{*}{\Rightarrow} w, \qquad (6.2)$$

with x, y in $(V \cup T)^*$. In words, a variable is useful if and only if it occurs in at least one derivation. A variable that is not useful is called **useless**. A production is useless if it involves any useless variable.

- A production is only useful if and only if there is at least one word that can be derived from a

variable

- If it is not useful it is useless!

- Secondly, a variable may become useless if there is no way of getting to that terminal string from it

□ Ex.

- S → A,
- A → aA | λ
- B → bA. //B can never be reached! → remove it!
  - › This can be changed to:
- S → A,
- A → aA | λ.

□ Ex. Eliminating all useless productions

- S → aS | A | C,
- A → a, //This is a useful production → Keep it
- B → aa. //This can never be reached → Remove it
- C → aCb. //this is an infinite loop → Remove it
  - › This can be updated to:
- S → aS | A,
- A → a.

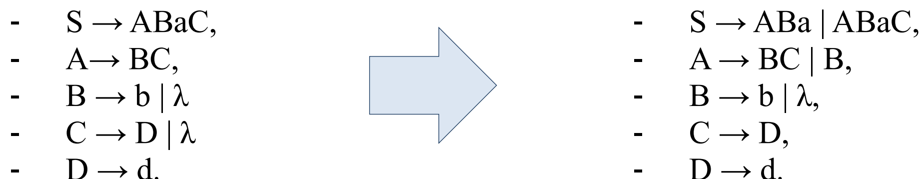
○ **Removing λ-productions**

- Any production of a CFG of the form: A → λ, is called a λ production. To remove nullable production we must find all variables that lead to the nullable production and change them

□ **Ex: S → ABaC, A → BC, B → b | λ, C → D | λ, D → d.**

› <u>Step 1. Remove λ production from C</u>

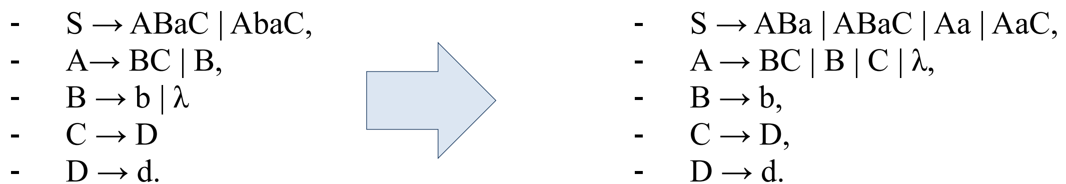
- S → ABaC,
- A→ BC,
- B → b | λ
- C → D | λ
- D → d.



- S → ABa | ABaC,
- A → BC | B,
- B → b | λ,
- C → D,
- D → d.

› <u>Step 2. Remove λ production from B</u>

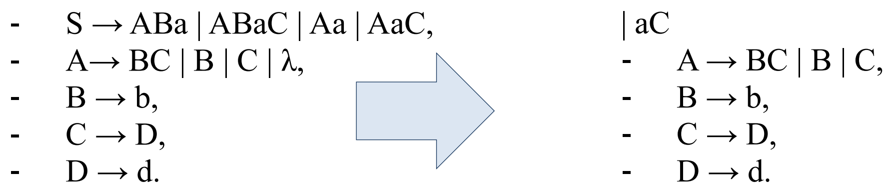
- S → ABaC | AbaC,
- A→ BC | B,
- B → b | λ
- C → D
- D → d.



- S → ABa | ABaC | Aa | AaC,
- A → BC | B | C | λ,
- B → b,
- C → D,
- D → d.

› <u>Step 3. Remove λ production from A</u>

- S → ABa | ABaC | Aa | AaC,
- A→ BC | B | C | λ,
- B → b,
- C → D,
- D → d.



| aC

- A → BC | B | C,
- B → b,
- C → D,
- D → d.

○ S → ABa | ABaC | Aa | AaC | Ba | BaC | a

Definition 6.3

Any production of a context-free grammar of the form

$$A \to B,$$

where $A, B \in V$, is called a **unit-production.**

- **Removing Unit Productions**
  - Unit production are production that produce only variables and have no elements
    - Ex.
      - S → Aa | B
      - B → A | bb,
      - A → a | bc | B.
        - First remove the unit production from → A
      - S → Aa | B
      - B → a | bc | B | bb,
      - A → a | bc | B.
        - Next remove all B → B productions and then unit production from → B
      - S → Aa | a | bc | bb
      - B → a | bc | bb,
      - A → a | bc | bb.
        - Finally remove useless (unreachable variables)
      - S → Aa | a | bc | bb,
      - A → a | bc | bb.
    - Steps to follow to remove undesirable productions
      1. Remove λ production
      2. Remove unit production
      3. Remove useless production

» 6.2 Two important Normal Forms

- **Chomsky Normal Form**
  - The number of symbols on the right is strictly limited to no more than 2
  - A CFG in Chomsky looks like this:
    - A → BC or A → a or S → Ab
  - **Ex.**
    - S → AS | a,
    - A → SA | b.
      - This is in Chomsky form!

- **Ex.**
  - $S \to ABa$,
  - $A \to aab$,
  - $B \to Ac$.
    - Change all elements to T_x and create production from $T_x \to x$, for each
  - $S \to ABT_a$,
  - $A \to T_aT_aT_b$,
  - $B \to AT_c$,
  - $T_a \to a$,
  - $T_b \to b$,
  - $T_c \to c$.
    - Now simplify each production to two variables by keeping the first one and creating a new one for the remaining variables
  - $S \to AV_1$
  - $A \to T_aV_2$,
  - $B \to AT_c$,
  - $V_1 \to BT_a$,
  - $V_2 \to T_aT_b$,
  - $T_a \to a$,
  - $T_b \to b$,
  - $T_c \to c$.

- **Greibach Normal Form**
  - Greibach puts a restriction on the position in which terminals and variables can appear
  - You must have 1 element per production and it must be the left most position
  - Ex.
    - $S \to AB$, //doesn't have any elements, must add 1
    - $A \to aA \mid bB \mid b$,
    - $B \to b$.
      - $S \to AB$, is not Grebach form, therefore you must replace A with its productions
    - $S \to aAB \mid bBB \mid bB$,
    - $A \to aA \mid bB \mid b$,
    - $B \to b$.
  - Ex.
    - $S \to abSb \mid aa$. //multiple elements per production and not left most
      - Replace right sided elements with variables
    - $S \to aBSB \mid aA$,
    - $A \to a$,
    - $B \to b$.

» **7.1 Nondeterministic pushdown Automata**

- CFGs can be either regular or irregular.
    - regular CFGs can be shown in a NFA or DFA, however CFGs that are not regular cannot be shown in an NFA or DFA.
        - Ex. $L = \{a^n b^n : n \geq 0\}$ // impossible to create an NFA, but easily create a CFG
            - $S \rightarrow aSb \mid \lambda$.
    - However with stacks, we can now use NFAs and DFAs to create a PDA which represents non regular CFG
- **Definition of a PDA**

Definition 7.1

A nondeterministic pushdown accepter (npda) is defined by the septuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, z, F),$$

where

Q is a finite set of internal states of the control unit,
Σ is the input alphabet,
Γ is a finite set of symbols called the **stack alphabet**,
$\delta : Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow$ set of finite subsets of $Q \times \Gamma^*$ is the transition function,
$q_0 \in Q$ is the initial state of the control unit,
$z \in \Gamma$ is the **stack start symbol**,
$F \subseteq Q$ is the set of final states.

- Similar to an NFA/DFA except that for each transition from one node to another, you must interact with the stack, either by adding something to the stack, or removing. Whatever you add, directs you to what you are allowed to remove
- $\delta(q_1, a, b) = \{(q_2, cd), (q_3, \lambda)\}$
    - this means if b is at the top of the stack, we either move from q_1 to q_2 adding 'a' to the string and replacing 'b' with 'cd' in the stack or we move from $q_1 \rightarrow q_3$ adding 'a' to the string and replacing 'b' with λ in the stack
    - in the case of 'cd', c will be at the top of the stack followed by d

- ### **The language Accepted by a pushdown Automaton**

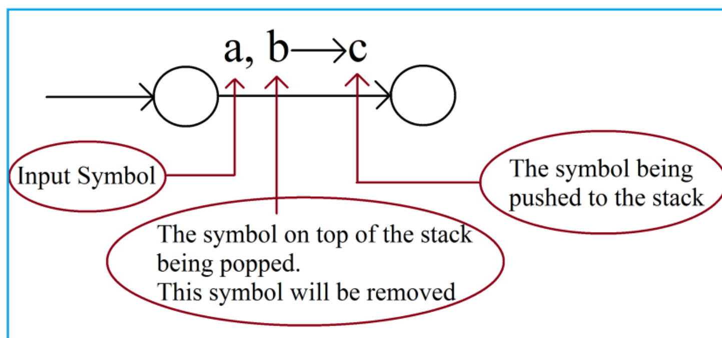
Definition 7.2

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ be a nondeterministic pushdown automaton. The language accepted by M is the set

$$L(M) = \left\{ w \in \Sigma^* : (q_0, w, z) \overset{*}{\vdash}_M (p, \lambda, u), p \in F, u \in \Gamma^* \right\}.$$

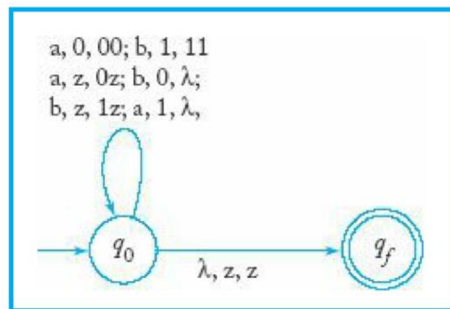
In words, the language accepted by M is the set of all strings that can put M into a final state at the end of the string. The final stack content u is irrelevant to this definition of acceptance.

- The language accepted by M is the set of all strings that can put M into a final state at the end of the string. The final stack content u is irrelevant to this definition of acceptance



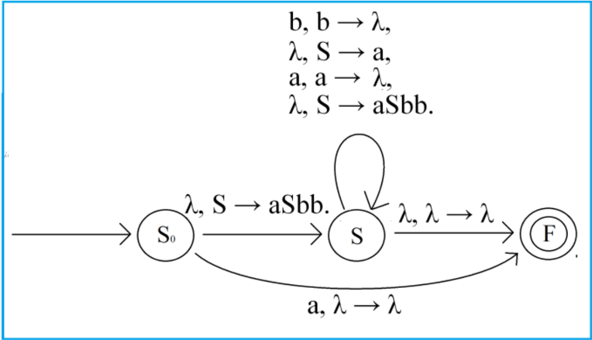
- the λ element can be used at any point to ignore the operation
  - You can add an element to the stack without adding a character: $\lambda, \lambda \rightarrow 1$
  - You can add a character without touching the stack: $a, \lambda \rightarrow \lambda$
  - You can add a character without popping the stack: $a. \lambda \rightarrow 1$
  - And so on.

□ Ex. $\{ w : n_a = n_b \}$ //number of a's = the number of b's



□ Ex. $\{ww^R\}$ //a palindrome with only even number of letters

» **7.2 Pushdown Automata and Context-Free languages**



- Every CFL there is a NPDA
    - **Pushdown automata for CFL**
        - Ex. Construct NPDA from:
            - S → aSbb | a
    - Context free grammars for pushdown automata
        - To keep things simple it helps to assume the npda in question meets two requirements
            1. It has a single final state qf that is entered if an only if the stack is empty
            2. Each move increases or decreases the stack content by a single symbol
        - While this may seem severe it actually does not affect the power of the pda
            - For any npda there exists an equivalent one having properties 1 and 2
        - If L = L(M) for some npda M, then L is a CFL

» **7.3 Deterministic pushdown automata and Determinisitic context free languages**

- A dpda is a pushdown automaton that never has a choice in its move
    - For any given input symbol and any stack top, at most one move can be made
    - When a λ-move is possible for some configuration, no input-consuming alternative is

Definition 7.3

A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ is said to be deterministic if it is an automaton as defined in Definition 7.1, subject to the restrictions that, for every $q \in Q$, $a \Sigma \cup \{\lambda\}$ and $b \in \Gamma$,

1. $\delta(q, a, b)$ contains at most one element,
2. if $\delta(q, \lambda, b)$ is not empty, then $\delta(q, c, b)$ must be empty for every $c \in \Sigma$.

The first of these conditions simply requires that for any given input symbol and any stack top, at most one move can be made. The second condition is that when a λ-move is possible for some configuration, no input-consuming alternative is available.

Definition 7.4

A language L is said to be a **deterministic context-free language** if and only if there exists a dpda M such that $L = L(M)$.

available

- How to find out whether a PDA is deterministic or not?
    - See if it has to make a choice. Any time a context free language encounters a choice it cannot be deterministic

» 8.1 Two Pumping Lemma

- An effective too for showing that certain languages are not regular
  - **A pumping lemma for Context free languages**
    - is useful in showing that a language does not belong to the family of context free languages
    - Unlike regular languages, CFL are bounded on |xyz|, making it more complicated
      - Theorem 8.1

Let L be an infinite context-free language. Then there exists some positive integer m such that any $\omega \varepsilon L$ with $|\omega| \geq m$ can be decomposed as

$$w = uvxyz, \quad (8.1)$$

with

$$|vxy| \leq m, \quad (8.2)$$

and

$$|vy| \geq 1, \quad (8.3)$$

such that

$$uv^i xy^i z \in L, \quad (8.4)$$

for all $i = 0, 1, 2, \ldots$. This is known as the pumping lemma for context-free languages.

Proof: Consider the language $L - \{\lambda\}$, and assume that we have for it a grammar G without unit-productions or λ-productions. Since the length of the string on the right side of any production is bounded, say by k, the length of the derivation of any $\omega \varepsilon L$ must be at least $|\omega|$/k. Therefore, since L is infinite, there exist arbitrarily long derivations and corresponding derivation trees of arbitrary height.

Consider now such a high derivation tree and some sufficiently long path from the root to a leaf. Since the number of variables in G is finite, there must be some variable that repeats on this path, as shown schematically in Figure 8.1. Corresponding to the derivation tree in Figure 8.1, we have the derivation

$$S \overset{*}{\Rightarrow} uAz \overset{*}{\Rightarrow} uvAyz \overset{*}{\Rightarrow} uvxyz,$$

where u, v, x, y, and z are all strings of terminals. From the above we see that and $A \overset{*}{\Rightarrow} vAy$and $A \overset{*}{\Rightarrow} x$, so all the strings uv^ixy^iz, $i=0,1,2$ can be generated by the grammar and are therefore in L. Furthermore, in the derivations $A \overset{*}{\Rightarrow} vAy$ and $A \overset{*}{\Rightarrow} x$,we can assume that no variable repeats. To see this, look at the sketch of the derivation tree in Figure 8.1. In the subtree T_5 no variable repeats; otherwise we could just apply the argument to this repeating variable. Similarly, we can assume that no variable repeats in the subtrees T_3 and T_4. Therefore, the lengths of the strings v, x, and y depend only on the productions of the grammar and can be bounded independently of w so that (8.2) holds. Finally, since there are no unit-productions and no λ-productions, v and y cannot both be empty strings, giving (8.3).

  - **A pumping lemma for linear languages**
    - We previously made a distinction between linear and nonlinear context free grammars, we

Definition 8.1

A context-free language L is said to be linear if there exists a linear context-free grammar G such that $L = L$ (G).

now make a similar distinction between languages

- Theorem 8.2

Let L be an infinite linear language. Then there exists some positive integer m, such that any $\omega \varepsilon L$ with $|\omega| \geq m$ can be decomposed as $w = uvxyz$ with

$$|uvyz| \leq m, \quad (8.5)$$
$$|vy| \geq 1, \quad (8.6)$$

such that

$$uv^i xy^i z \in L, \quad (8.7)$$

for all i =0, 1, 2,….

Note that the conclusions of this theorem differ from those of Theorem 8.1, since (8.2) is replaced by (8.5). This implies that the strings v and y to be pumped must now be located within m symbols of the left and right ends of w, respectively. The middle string x can be of arbitrary length.

Proof: Since the language is linear there exists a linear grammar G for it. For the argument it is convenient to assume that G has no unit-productions and no λ-productions. An examination of the proofs of Theorem 6.3 and 6.4 makes it clear that removing unit-productions and λ-productions does not destroy the linearity of the grammar. We can therefore assume that G has the required property.

Consider now the derivation of a string $\omega \varepsilon L(G)$

$$S \stackrel{*}{\Rightarrow} uAz \stackrel{*}{\Rightarrow} uvAyz \stackrel{*}{\Rightarrow} uvxyz = w.$$

Assume, for the moment, that for every w G L(G), there is a variable A, such that

1. in the partial derivation $S \stackrel{*}{\Rightarrow} uAz$ no variable is repeated,
2. in the partial derivation $S \stackrel{*}{\Rightarrow} uAz \stackrel{*}{\Rightarrow} uvAyz$ no variable except A is repeated,
3. the repetition of A must occur in the first m steps, where m can depend on the grammar, but not on ω.

If this is true, then the lengths of u, v, y, z must be bounded independent of w. This in turn implies that (8.5), (8.6), and (8.7) must hold.

(a) $L = \{a^n b^j : n \leq j^2\}$.

» **8.2 Closure properties and decision algorithm for Context Free Languages**

- **Languages**
  - Closure properties that hold for regular languages do not always hold for context free languages
- **Closer of Context free Languages**
  - The family of context free languages is closed under union, concatenation and star closure
  - The family of context free languages is NOT closed under intersection and complementation
  - Let L_1 be a context free language and L_2 be a regular languages, then $L_1 \cap L_2$ is context free

Chapter 9: Turing Machines

- So far we have focused on the concept of regular and context free languages and their association with finite automata and pushdown accepters
- Regular languages form a proper subset of the Context-free languages and, therefore, pushdown automata are more powerful than finite automata.
  - Regular Languages = NFA, DFA, NPDA, DPDA
  - Context free = NPDA, DPDA
- We also saw that context free languages, while fundamental to the study of programming languages, are limited in scope
  - $L_1 = a^n b^n c^n$ and $L_2 = ww$ are two ex of simple languages that aren't context free
- This is why we decide to look beyond Context Free Languages
  - Create new families to include more languages
- Difference between NFA and an NPDA is the stack (ability to store a value)
- What about using two stacks? Or how do we create the most powerful automata → leads us to the Turing Machine

» **9.1 The Standard Turing Machine**

- Turing machine's storage is actually quite simple and viasualized as a single one dimensional array of cells. Rather than a stack that you can only access the top element, with an array we will be able to access any element
  - **Definition of a Turing machine**
    - It is an automaton whose temporary storage is a tape. This tape is divided into cells, each of which is capable of holding one symbol
    - Associate with this tape is a read write head that can travel right or left throughout this tape. Think of it as a pointer and different symbol in the tape
    - We assume that the input alphabet is a subset of the tape alphabet, not including the blanks
    - Blanks are ruled out as input

Definition 9.1

A Turing machine M is defined by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F),$$

where

Q	is the set of internal states,
Σ	is the input alphabet
Γ	is the finite set of symbols called the **tape alphabet**,
δ	is the transition function,
$\square \in \Gamma$	is a special symbol called the **blank**,
$q_0 \in Q$	is the initial state,
$F \subseteq Q$	is the set of final states.

- In figure 9.2 we have an example of a movement
  - Starting at node q0, if the next letter in the tape is an “a”, move to q1, change the a to a “d” and move right in the tape

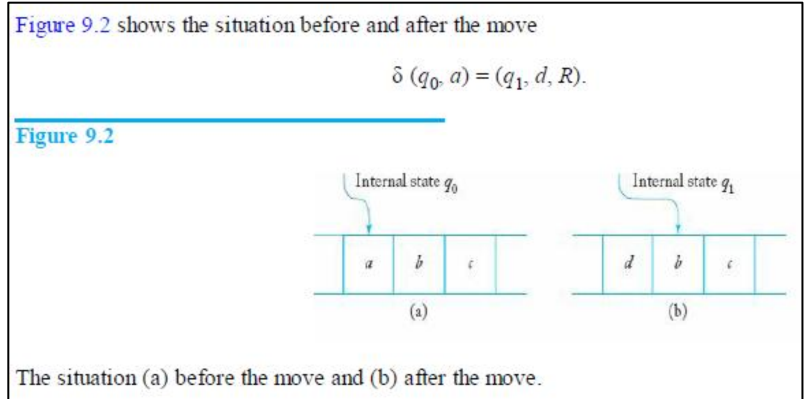


Figure 9.2 shows the situation before and after the move

$$\delta(q_0, a) = (q_1, d, R).$$

Figure 9.2

The situation (a) before the move and (b) after the move.

» **9.1 The Standard Turing Machine**

○ **Definition of a Turing machine**

- The Turing machine starts at the initial state and then moves through the tape using the transition functions from each node.
- As you can see in figure 9.4, the transition moves are very simple in the automaton.
  › a,b,R; this means if we are currently on this node, and an a shows up, than change it to a b, move Right on the tape and stay in q0.
  › You can only move to q1, once the tape reaches a blank.

Example 9.2

Consider the Turing machine defined by

$$Q = \{q_0, q_1\},$$
$$\Sigma = \{a, b\},$$
$$\Gamma = \{a, b, \square\},$$
$$F = \{q_1\},$$

and

$$\delta(q_0, a) = (q_0, b, R),$$
$$\delta(q_0, b) = (q_0, b, R),$$
$$\delta(q_0, \square) = (q_1, \square, L),$$

If this Turing machine is started in state q_0 with the symbol a under the read-write head, the applicable transition rule is $\delta(q_0,a) = (q_0,b,R)$. Therefore, the read-write head will replace the a with a b, then move right on the tape. The machine will remain in state q_0. Any subsequent a will also be replaced with a b, but b's will not be modified. When the machine encounters the first blank, it will move left one cell, then halt in final state q_1.

Figure 9.3 shows several stages of the process for a simple initial configuration.

q_0 | a | a | ; q_0 | b | a | ; q_0 | b | b | ; q_1 | b | b |

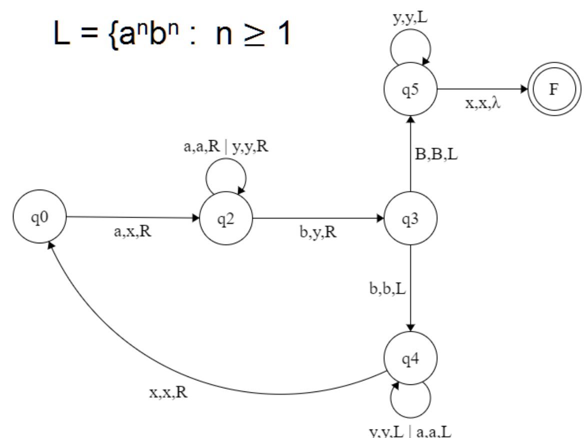
A sequence of moves.

As before, we can use transition graphs to represent Turing machines. Now we label the edges of the graph with three items: the current tape symbol, the symbol that replaces it, and the direction in which the read-write head is to move. The Turing machine in Example 9.2 is represented by the transition graph in Figure 9.4.

Figure 9.4

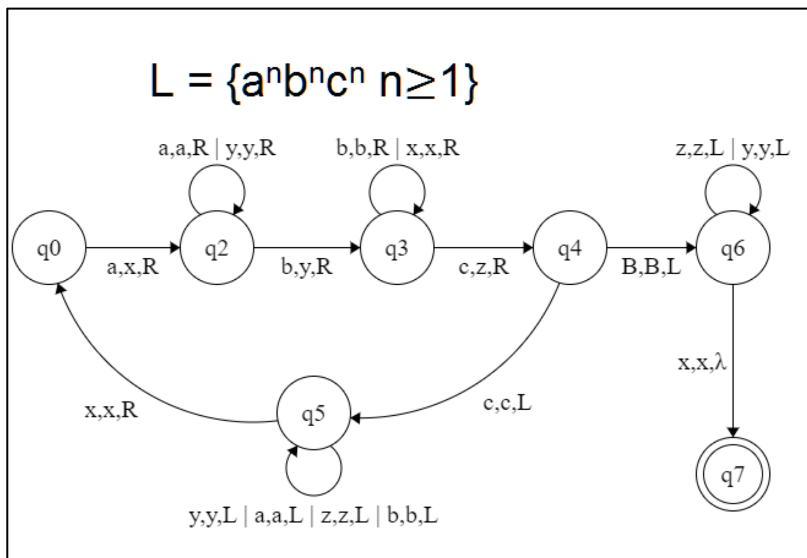
a, b, R;
b, b, R
q_0 —□,□, L→ q_1

- **Summarizing key functions**
  1. The Turing machine has a tape that is unbounded in both directions, allowing any number of left and right moves
  2. The Turing machine is deterministic in the sense that the transition defines at most one move for each configuration
  3. There is no special input file. We assume that at the initial time the tape has some specified content. Some of this may be considered input. Similarly, there is no special output device. Whenever the machine halts, some or all of the contents of the tape may be viewed as output
  - Instantaneous description: Any configuration of the Turing machine has three requirements
    1. the state the Turing Machine is in
    2. The contents of the tape
    3. The position of the tape head on the tape
- **Turing Machines a Languages Accepters**
  - How can a TM accept a string? → write the string into the tape, with blanks filling out the unused portions
    - Machine starts in the initial state q0 with the read write head positioned on the leftmost symbol of the word
    - If after a sequence of moves, the Turing machine enters a final state and halts, then w is considered to be accept
  - This is why we exclude blanks when inputted a string → the machine wouldn't be able to distinguish between the blanks inputted and the blanks that indicate the start and end of the input
  - If the word doesn't exist in L(M), there are two cases that could happen,
    1. It enters an infinite loop and halts
    2. It halts on a state that isn't a final state
  - Complicated languages becomes more difficult because of the primitive instruction set
  - Example: →
    - As you can see, something that can easily be done with a PDA becomes a bit more complicated with a Turing Machine

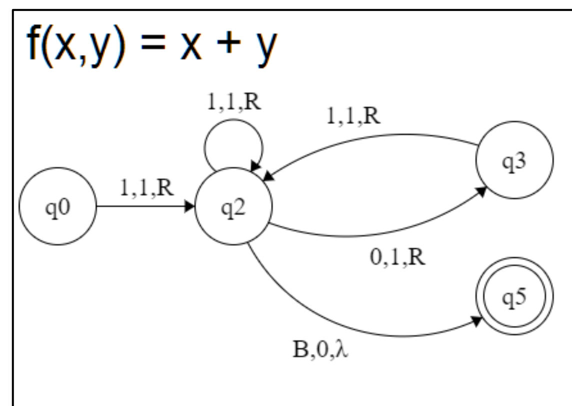


» **9.1 The Standard Turing Machine**

- **Turing Machines a Languages Accepters**
  - Example
    - Similar to the last one, but as you can see when we reach the blank at the end of the c's, we then go back to see that we only have y's and then we hit an x
    - This means that we have crossed off all the a's, b's and c's equal times

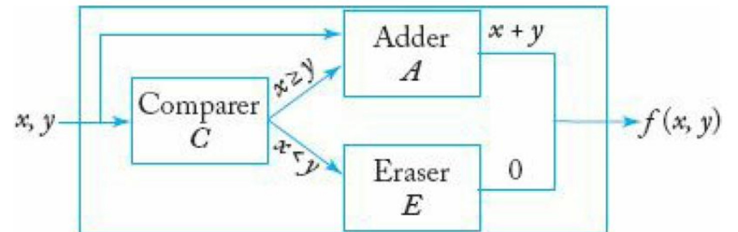


  - One conclusion we can draw form this example is that a Turing Machine can recognize some languages that are not context free, a first indication that Turing Machines are more powerful than Push down automata
- **Turing Machines as Transducers**
  - Turing machines are not only good for language accepters but they also provide us with simple abstract model for digital computers in general
  - Because of the transducers: changing input to create an output, we can now use it to build functions
  - All common mathematical functions, no matter how complicated are Turing-computable
  - Given two positive intergers x and y, design a Turing machine that computers x + y
    - Representation: unary form → 5 = xxxxx, or $|w(x)| = x$
    - Then decide how to separate x and y
      - › → we can use a 0
    - Now we have a formula (x + y)
      - › → B 1^x 0 1^y B
      - › 5 + 3 → B111110111B
    - Object now? To get all the ones together at the beginning

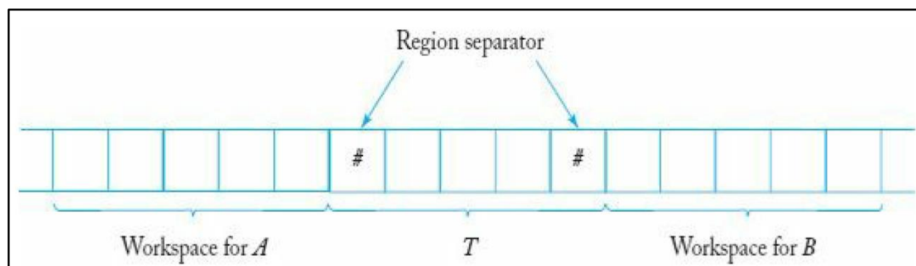


» **9.2 Combining Turing Machines for Complicate Tasks**

- Simple building blocks to create complex instructions
- How to combine many simple functions to create a complex program
  - Break it into each part and connect
- View an overview Example: f(x,y) = x + y if $x \geq y$ | 0 if $x < y$
  - Representation unary: we can represent 0 as B0B //blanks with a 0 in the middle
  - Split the problem into two stages
    - 1st stage: evaluate if $x \geq y$
    - 2nd stage: complete the appropriate function



  - The diagram on the right explain the 2 stages
    - When the comparer evaluates the expression, whichever one is correct, a 1 will be sent to that adder(or erase). That 1 will be used in the Boolean expression to complete the operations and return an answer
- A Turing machine can be used as a subprogram that can be invoked repeatedly by another Turing Machine //This requires storage
  - Start machine A, which then invokes B, and at the end of B, we continue with A, therefore we have to store the information of A, while we go through the machine of B
  - To solve this we divided the tape into regions (workspaces) some for A, some for B, or however many process the overall function needs



  - This is very intuitive. When A needs to call B, it has a loop that loops it through all characters till it reaches # separator, at that moment it has left the memory of A in the work space and now is pointing to an area that can be reserved for input for B, the following # separator will designate the start of the B tape, B takes information from T, and then outputs it back to T, and then moves back to A to complete the operation

» **9.3 Turing's Thesis**

- Turing machine are easy in theory hard in practice
- we know Turing Machine's are more powerful than PDAs
- we know TM can do simple operations, simple comparisons, string manipulations, and can be combined to form more complex machines
- While it is almost impossible to prove that a TM can solve any problem a computer program can solve, there also isn't any proof that it can't
- Lead to this statement: every indication is that TM are in principle as powerful as any computer
    - Any computation that can be carried out by mechanical means can be performed by some Turing machine
    - Cannot be proved, because of the lack of definition of "mechanical means"
- Arguments for Accepting this Turing Thesis
    1. Anything that can be done on any existing digital computer can also be done by a Turing Machine
    2. No one has yet been able to suggest a problem, solvable by what we intuitively consider an algorithm, for which a Turing Machine program cannot be written
    3. Alternative models have been proposed for mechanical computation, but none of them is more powerful than the Turing machine model

Chapter 10: Other Models of Turing Machines

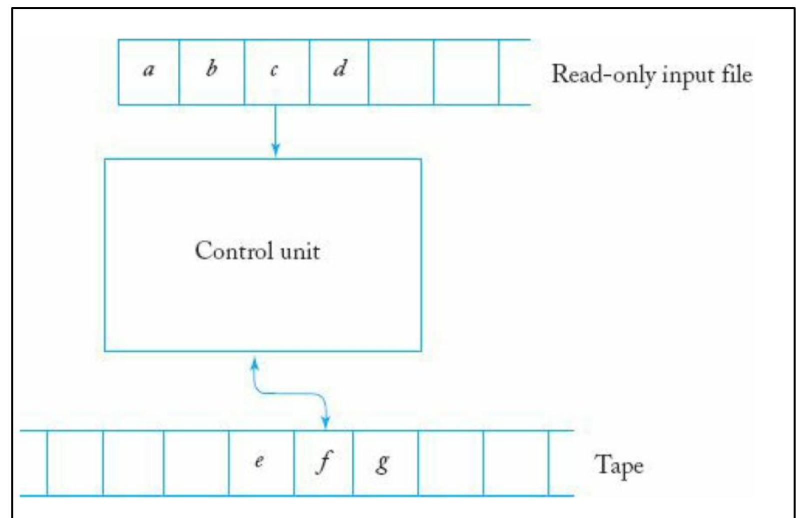
» **<u>Introduction</u>**

- Other Turing Machine models that can be equally as strong
  - Turing Machines with more than one tape
  - Non-deterministic Turing Machines
  - Stay option Turing machines
  - Semi-bounded Turing machines
- None of them have any more power than a regular TM

» **<u>10.1 Minor Variations on the Turing machine them</u>**

- **Equivalence of class of automata**
  - Two automata are equivalent if they accept the same language
  - Consider two classes of automata C_1 and C_2: every M_1 in C_1 there is an automaton M_2 in C_2 such that: $L(M_1) = L(M_2)$
    - And that the reverse is true → equivalent
    - If only one of those two conditions hold than one is more powerful than the other
- **Turing Machines with a stay option**
  - In the standard definition the read-write head must move either to the right or to the left, sometimes it would be convenient to provide a third option, to have the read-write head to stay in place after rewriting the cell content
    - For this option we have three (L,R,S) //S = stay
  - It is clear that every Standard TM can be implemented by one with a stay option, what about the other way? Can all stay option TM be implement by a standard TM??
    - The answer is yes! Although it will be more complicated and have more nodes, it will be able to create a loop that will return back to it without effecting anything else
    - Two step process 1,0,R → 1,1L | 0,0, L | B, B, L //no matter the element it will return back to the previous node with no effect. Returns that exact same as 1,0,S
- **Turing Machine with Semi-Infinite tape**
  - Many people believe the standard TM have a tape that is unbounded in only one direction (to the right)
    - Just imagine the you are limited in the amount of left moves you can make
    - Same powers as an infinite tape
  - For our purpose we believe the standard TM is unbounded on both ends

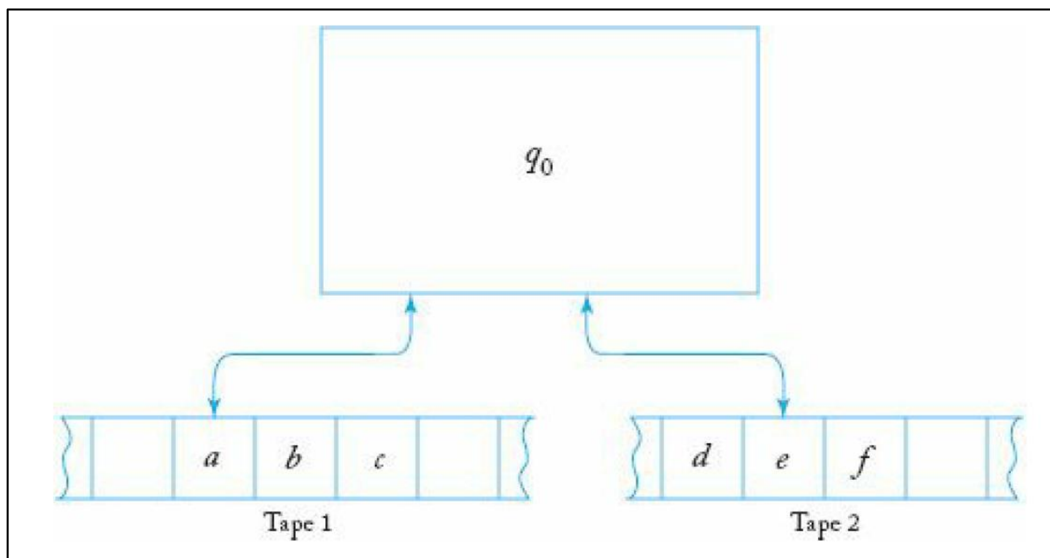
- **The Offline Turing Machine**
  - Has a separate input stream to the tape
  - Is to see that you can do everything with this machine as you would with a standard TM since you can just copy the entire input into the Tape before you begin



» **<u>10.2 Turing Machines with more complex storage</u>**

- **Multitape Turing Machines**
  - One with several tapes, each with its own independently controlled read write head
    - Transitions are updated to consider the added tape(s)

$$\delta\ (q_0, a, e) = (q_1, x, y, L, R)$$



    - From the transition, from q0 to q1, if the pointer of tape1 is pointing at a and the pointer of tape2 is pointing at an e, then they are change to x, y and moved left and right, respectively
  - A machine like this clearly gives more options at each state, but all it does is reduce the amount of states needed rather than increase the power of the TM
- **Multidimensional Turing machines**
  - One with a tape that is like a two dimensional array

» **<u>10.3 Nondeterministic Turing Machine</u>**

- Like any other non-deterministic accepter, you can have a choice at a state to make two separate moves
- Same power as a deterministic TM

» **10.4 a Universal Turing Machine**

- Reprogrammable Turing machine
  - A Turing machine that holds all machines at the same time
  - You give it an input and which machine to you, and it applies that input to that machine
- how this works?
  - Each TM is represented by a string of 0s and 1s
  -