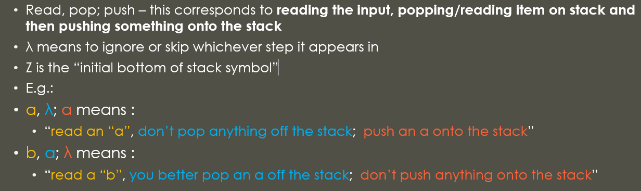
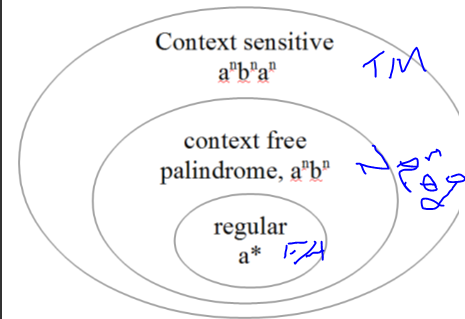
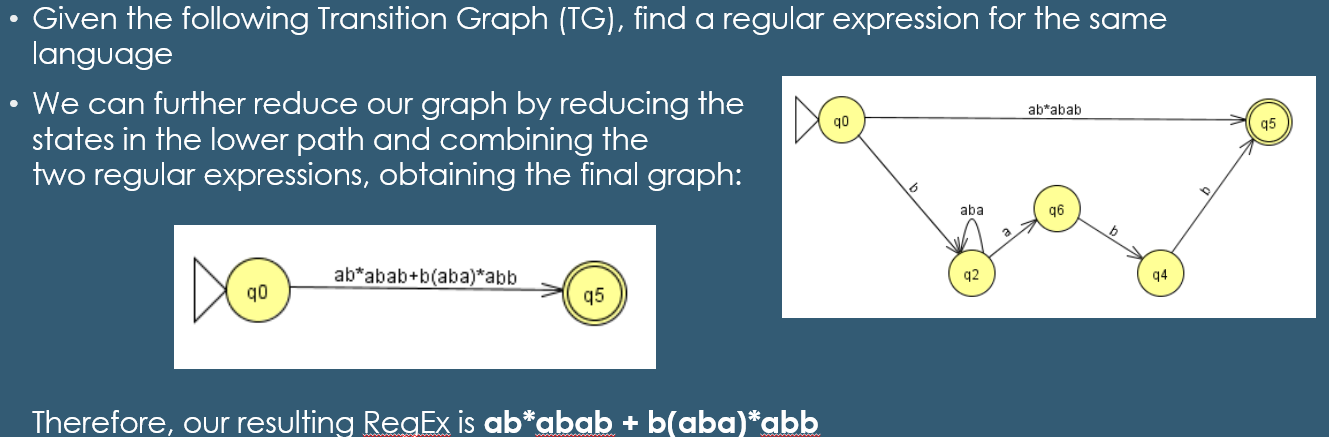
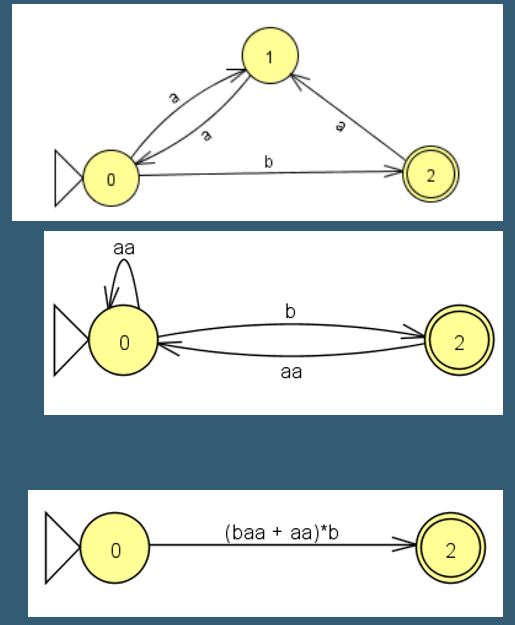
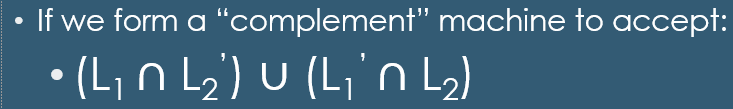
* In formal language theory, we ha ve four primary ways we use to model languages
  1. By describing them (Description)
  2. By recursive definition (Recursion)
  3. By using regular expressions
  4. By using finite automata
* As review, a **grammar** consists of a collection of
  1. Terminals, Σ
  2. Non-terminals (also called Variables), Γ
  3. Productions
     + Also called substitution rules, or just rules
* One variable is considered the **start variable**
* **And again**, formally, a **context free grammar** is a grammar where all the productions are of the form:
  1. Non-Terminal 🡪 *finite string of terminals and non-terminals*
* A **pushdown automaton** is a collection of four things:
  1. A finite alphabet Σ of input characters
  2. A finite set of states Ω at least one of which is denoted as a start state
  3. A finite set of stack characters Π (one of which is Z)
  4. A finite set of transitions between the states of the form **read, pop; push**
* The following is a definition of the language PEI (positive, even, integers)
* PEI = {2, 4, 6, 8, …}
* Recursive definition:
  1. Base case: 2 ∈ PEI
  2. Recursive step: If x ∈ PEI, so is x + 2
  3. Nothing else : nothing not described in (1) or (2) is in PEI
* All other regular expressions are formed using the above rules, and that all regular expressions are special strings (words) that use the letters of Σ and the five symbols:
* ( ) \* + λ
* Consider the language L, defined over the alphabet Σ = {a, b}
* Ex. 1: Give a regular expression to generate words having at least one a
  1. (a+b)\*a(a+b)\*
* A finite automaton takes a string as input, and gives output of “yes” or “no”, indicating whether the input string is part of the language
* Definition 1: A finite automaton consists of three things:
  1. A finite number of states (denoted σ), one of which is denoted the start state (typically σ0), and some (or none) of which are designated as the **final** or **accept** state(s.)
  2. A finite alphabet (denoted Σ) of possible input letters (symbols.)
     + We will typically use Σ= {a, b} for the sake of simplicity
  3. A finite set of rules or transitions, T, that uniquely map every state and every letter into another state, in other words, T : σ × Σ 🡪 σ
* Note that there can be more than 1 accept state but only 1 start state for FA.
  1. Make Transition Diagrams based on FA
* A **transition graph** relaxes the requirements on the FA
  1. Allows multiple start states
  2. Allows words or even the empty string, λ on an edge
  3. Allows 0 or 1 or any number of edges from a state

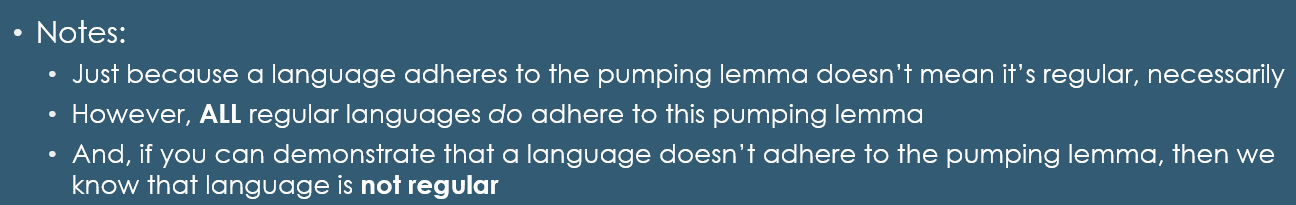
TG:

* We now have non-determinism
* Therefore, the machine can crash!
* Kleene’s Theorem states that if a language L can be defined by either
  + Regular Expression
  + Finite Automaton
  + Transition Graph
* Then it can be defined by ***all three***!
* We will develop a proof in four parts, with examples in each (where applicable or necessary for our purposes):
  + Given a Finite Automaton (FA), find a Transition Graph (TG), that is FA => TG
  + Given a Transition Graph (TG), find a Finite Automaton (FG), that is TG => FA
  + Given a Transition Graph (TG), find a Regular Expression (RE), that is TG => RE
  + Given a Regular Expression (RG), find a Transition Graph (TG), that is RE => TG
* It should be fairly clear that if we can demonstrate that the above four parts hold, then Kleene’s Theorem is proven
* What do we do in our FA if in the TG there are multiple start states?
  + Use a group state as a start state
  + Collapse lambda edges
  + TG Crashes 🡪 make a black hole for FA
  + We expand the string into intermediate state
  + The convention used in the book is to use **StartState**.EndStateLetterPosition
  + Thus, if we’re in state 0, on our way to state 1, and got an “a” (position 1 letter), then the intermediate state is 0.11
  + Note this only works up to 9 end states and/or letter positions
  + We could use a different convention, such as Start.End.LetterPos
* Proof part 3: TG 🡪 RE





Decidability

* Regular languages are only one category of formal languages
  + In fact, they’re often some of the most simple
* Recall that regular languages:
  + Can be described by a regular expression (or accepted by a FA)
* All **finite languages** are regular
* Consider an infinite (regular) language L
  + By definition, L must be accepted by a FA, *M*
  + The FA, *M* must have a finite number of states (by definition of a FA)
  + However, since L is infinite, there must be strings in L which are as long as we please, and thus *M* contains strings with more symbols than there are states in *M*
  + Thus, since M accepts every string in L, there must be a loop in M
* Clearly, if a FA has N states, and accepts strings of length N or greater, it will have to pass through at least one state more than once in order to accept these strings
* So, given a string with length N or greater, we can construct even longer strings by repeating (pumping) a given substring over and over again
* **Formally, the Pumping Lemma states:**
* If L is a regular language, then there is a number, p (the pumping length), where, if w is any string (word) in L of at length at least p, then w may be divided into three pieces:
  + w = xyiz
    - **The book is wrong (p. 84) !!! (it says xyzi – that’s incorrect)**
* And, the following conditions must be satisfied:
  + For each i ≥ 0, xyiz ∈ L
  + y ≠ λ (in other words, |y| ≥ 1, or, y must be at least length 1)
  + |xy| ≤ p
* **Note/Reminder:** x, y, and z are *strings -* not necessarily single letters!!!!