Computer Language Theory

Chapter 2: Context-Free Languages

Last modified 3/4/21

Overview

- In Chapter 1 we introduced two equivalent methods for describing a language: Finite Automata and Regular Expressions
- In this chapter we do something analogous
  - We introduce context free grammars (CFGs)
  - We introduce push-down automata (PDA)
    - PDAs recognize CFGs
    - In my view the order is reversed from before since the PDA is introduced second
  - We even have another pumping lemma (Yeah!)

Why Context Free Grammars

- They were first used to study human languages
  - You may have even seen something like them before
- They are definitely used for "real" computer languages (C, C++, etc.)
  - They define the language
  - A parser uses the grammar to parse the input
  - Of course you can also parse English

Section 2.1

Context-Free Grammars

A Context-Free Grammar

- Here is an example grammar G1

  A → 0A1

  A → B

  B → #

- A grammar has substitution rules or productions
  - Each rule has a variable and arrow and a combination of variables and terminal symbols
    - We will capitalize symbols but not terminals
  - A special variable is the start variable
    - Usually on the left-hand side of topmost rule
  - Here the variables are A and B and the terminals are 0, 1, #

Using the Grammar

- Use grammar to generate a language by replacing variables using rules in the grammar
  - Begin with the start variable (in this case "A")
  - Give me some strings grammar G1 below generates?

$A \rightarrow 0A1$

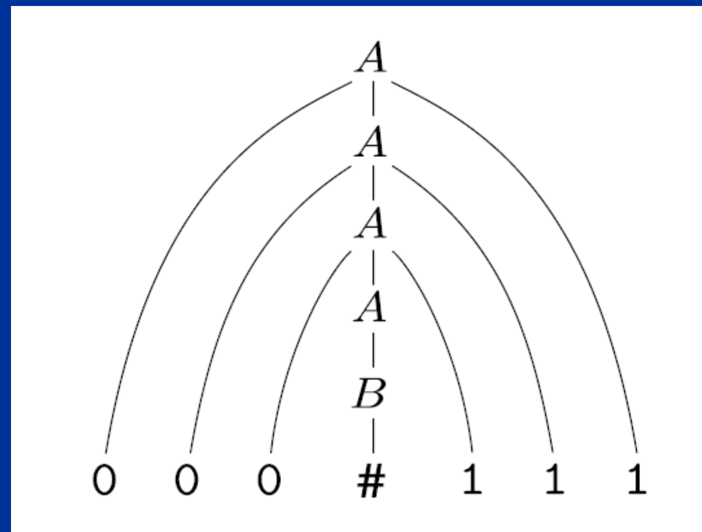
$A \rightarrow B$

$B \rightarrow \#$

    - One answer: 0#1, 00#11, 000#111, …
    - The sequence of steps is the derivation
    - For this example the derivation is:
      - $A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$

Parse Trees

- Instead of listing the derivation you can provide a parse tree, which is more visual. The parse tree for generating 000#111 is below.



The Language of Grammar G1

- All strings generated by G1 form the language
  - We write it L(G1)
  - What is the language of G1?
    - $L(G1) = \{0^n\#1^n \mid n \geq 0\}$
  - Can we generate L(G1) with a FA?
    - No. We proved that the language 0^n1^n is not regular
    - We just showed something significant
      - CFGs can generate non-regular languages
      - But we have not yet shown they can generate all regular languages

An Example English Grammar

- Page 101 of the text has a simplified English grammar
  - Follow the derivation for “a boy sees”
  - Can you do this without looking at the solution?
  - Having a good knowledge of English grammar would help

Formal Definition of a CFG

- A CFG is a 4-tuple (V, Σ, R, S) where
  1. V is a finite set called the variables
  2. Σ is a finite set, disjoint from V, called the terminals
  3. R is a finite set of rules, with each rule being a variable and a string of variables and terminals, and
  4. $S \in V$ is the start variable

Example

- Grammar G3 = ({S}, {a,b}, R, S), where:

  $S \rightarrow aSb \mid SS \mid \varepsilon$

  - What does this generate:
    - abab, aaabbb, aababb
    - If you view a as "(" and b as ")" then you get all strings of properly nested parentheses
      - Note they consider ()() to be okay
      - Key property: at any point will have at least as many a's as b's
        - Ultimately number of a's must equal number of b's
  - Generate the derivation for aababb
    - $S \rightarrow aSb \rightarrow aSSb \rightarrow aaSbSb \rightarrow aabSb \rightarrow aabaSbb \rightarrow aababb$

Example 2.4 Page 103 (2nd ed)

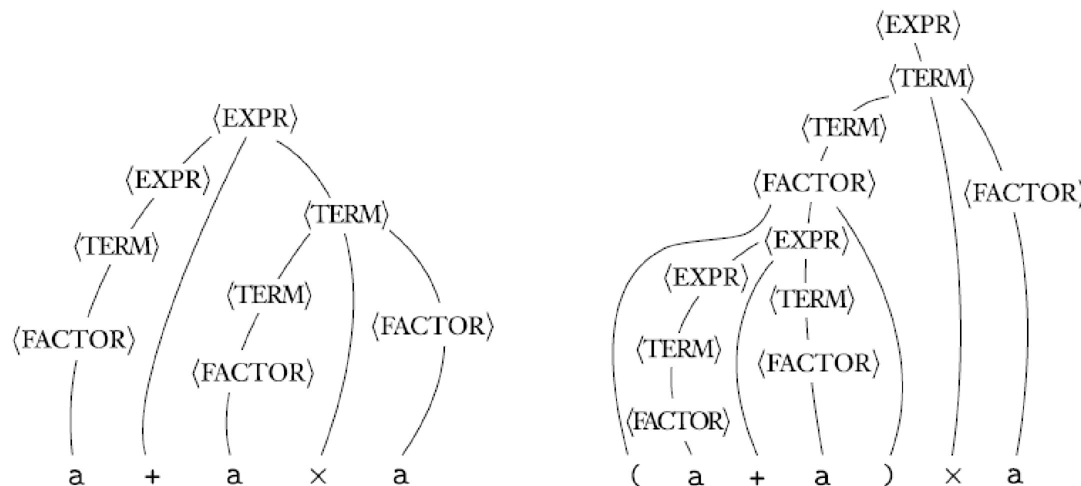
Parse tree determines order of operations not algebraic rules. Even if parentheses were omitted from grammar, the 2nd parse tree would add a+a before multiplying by a

Consider grammar $G_4 = (V, \Sigma, R, \langle \text{EXPR} \rangle)$.

V is $\{\langle \text{EXPR} \rangle, \langle \text{TERM} \rangle, \langle \text{FACTOR} \rangle\}$ and Σ is $\{\texttt{a}, \texttt{+}, \texttt{x}, \texttt{(}, \texttt{)}\}$. The rules are

$$\langle \text{EXPR} \rangle \rightarrow \langle \text{EXPR} \rangle \texttt{+} \langle \text{TERM} \rangle \mid \langle \text{TERM} \rangle$$
$$\langle \text{TERM} \rangle \rightarrow \langle \text{TERM} \rangle \texttt{x} \langle \text{FACTOR} \rangle \mid \langle \text{FACTOR} \rangle$$
$$\langle \text{FACTOR} \rangle \rightarrow \texttt{(} \langle \text{EXPR} \rangle \texttt{)} \mid \texttt{a}$$

The two strings a+axa and (a+a)xa can be generated with grammar G_4. The parse trees are shown in the following figure.



Designing CFGs

- Like designing FA, some creativity is required
  - Probably harder with CFGs since are more expressive than FA (we will show that soon)
  - Here are some guidelines
    - If the CFL is the union of simpler CFLs, design grammars for the simpler ones and then combine
      - For example, $S \rightarrow G1 \mid G2 \mid G3$
    - If the language is regular, then can design a CFG that mimics a DFA
      - Make a variable Ri for every state qi
      - If $\delta(qi, a) = qj$, then add $Ri \rightarrow aRj$
      - Add $Ri \rightarrow \varepsilon$ if i is an accept state
      - Make R0 the start variable where q0 is the start state of the DFA
    - Assuming this really works, what did we just show?
      - We showed that CFGs subsume regular languages

Designing CFGs continued

- If you are asked to design a CFL with some property, establish that property at the start, and then generate all possible string that maintain the property
  - HW example: {w| w contains at least three 1s}
    - Start with a string with 3 1's and then generate all possible strings keeping those 3 1's
      - Generate something that represents $\Sigma^*1\Sigma^*1\Sigma^*1\Sigma^*$
        - $S \rightarrow R1R1R1R$
        - $R \rightarrow 0R|1R|\varepsilon$

Designing CFGs continued

- A final guideline:
  - Certain CFLs contain strings that are linked in the sense that a machine for recognizing this language would need to remember an unbounded amount of information about one substring to “verify” the other substring.
    - This is sometimes trivial with a CFG
    - Example: 0^n1^n
      - $S \rightarrow 0S1 \mid \varepsilon$

Ambiguity

- Sometimes a grammar can generate the same string in multiple ways
  - If a grammar generates even a single string in multiple ways then the grammar is *ambiguous*
  - Example 2.4 from a few slides ago was ambiguous:

    $\text{EXPR} \rightarrow \text{EXPR} + \text{EXPR} \mid \text{EXPR} \times \text{EXPR} \mid (\text{EXPR}) \mid \text{a}$

    - This generates the string $a+a \times a$ ambiguously via 2 different parse trees

An English Example

- Grammar G2 on page 101 ambiguously generates *the girl touches the boy with the flower*
- Using your extensive knowledge of English, what are the two meanings of this phrase

Definition of Ambiguity

- A grammar generates a string ambiguously if there are two different parse trees
  - Two derivations may differ in the order that the rules are applied, but if they generate the same parse tree, it is not really ambiguous
  - Definitions:
    - A derivation is a <u>leftmost derivation</u> if at every step the leftmost remaining variable is replaced
    - A string *w* is derived <u>ambiguously</u> in a CFG G if it has two or more different leftmost derivations.

Chomsky Normal Form

- It is often convenient to convert a CFG into a simplified form
- A CFG is in Chomsky normal form if every rule is of the form:

  $A \rightarrow BC$

  $A \rightarrow a$

  Where a is any terminal and A, B, and C are any variables– except B and C may not be the start variable. The start variable can also go to ε

  - Any CFL can be generated by a CFG in Chomsky normal form

Converting CFG to Chomsky Normal Form

- Here are the steps:
  - Add rule $S_0 \rightarrow S$, where S was original start variable
  - Remove ε-rules. Remove $A \rightarrow \varepsilon$ and for each occurrence of A add a new rule with A deleted.
    - If we have $R \rightarrow uAvAw$, we get:
      - $R \rightarrow uvAw \mid uAvw \mid uvw$
  - Handle all unit rules
    - If we had $A \rightarrow B$, then whenever a rule $B \rightarrow u$ exists, we add $A \rightarrow u$.
  - Replace rules $A \rightarrow u_1 u_2 u_3 \ldots u_k$ with:
    - $A \rightarrow u_1 A_1$, $A_1 \rightarrow u_2 A_2$, $A_2 \rightarrow u_3 A_3 \ldots A_{k-2} \rightarrow u_{k-1} u_k$
  - You will have a HW question like this
    - Prior to doing it, go over example 2.10 in the textbook (page 108)

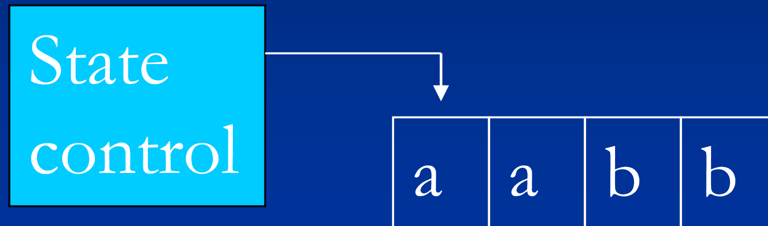
Section 2.2

Pushdown Automata

Pushdown Automata (PDA)

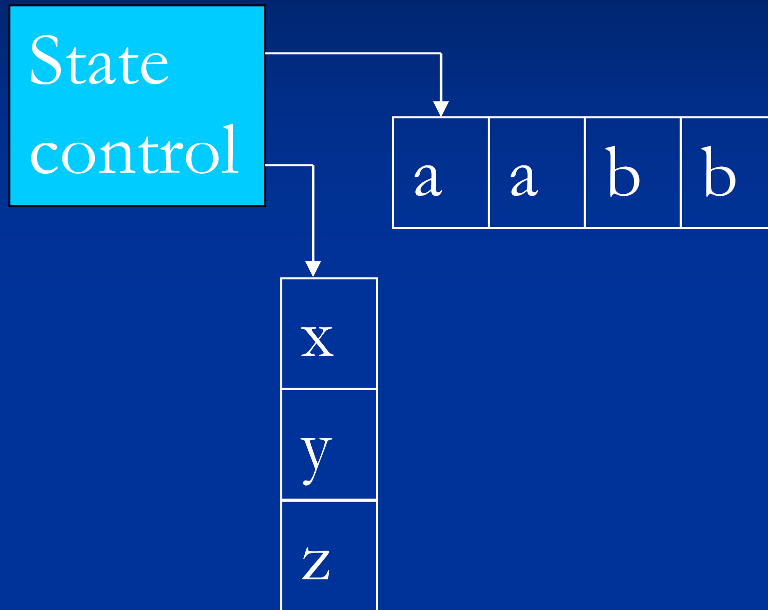
- Similar to NFAs but have an extra component called a *stack*
  - The stack provides extra memory that is separate from the control
- Allows PDA to recognize non-regular languages
- Equivalent in power/expressiveness to a CFG
- Some languages easily described by generators others by recognizers
- Nondeterministic PDA's not equivalent to deterministic ones but NPDA = CFG

Schematic of a FA



- The state control represents the states and transition function
- Tape contains the input string
- Arrow represents the input head and points to the next symbol to be read

Schematic of a PDA



- The PDA adds a stack
  - Can write to the stack and read them back later
  - Write to the top (push) and rest "push down" or
  - Can remove from the top (pop) and other symbols move up
  - A stack is a LIFO (Last In First Out) and size is not bounded

PDA and Language 0^n1^n

- Can a PDA recognize this?
  - Yes, because size of stack is not bounded
  - Describe the PDA that recognizes this language
    - Read symbols from input. Push each 0 onto the stack.
    - As soon as a 1's are seen, starting popping one 0 for each 1
    - If finish reading the input and have no 0's on stack, then accept the input string
    - If stack is empty and 1s remain or if stack becomes empty and still 1's in string, reject
    - If at any time see a 0 after seeing a 1, then reject

Formal Definition of a PDA

- The formal definition of a PDA is similar to that of a FA but now we have a stack
  - Stack alphabet may be different from input alphabet
    - Stack alphabet represented by Γ
  - Transition function key part of definition
    - Domain of transition function is $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon$
    - The current state, next input symbol and top stack symbol determine the next move

Definition of PDA

- A pushdown automata is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are finite sets
  1. Q is the set of states
  2. Σ is the input alphabet
  3. Γ is the stack alphabet
  4. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow P(Q \times \Gamma_\varepsilon)$ is transition function
  5. $q_0 \in Q$ is the start state, and
  6. $F \subseteq Q$ is the set of accept states
- Note that at any step the PDA may enter a new state and possibly write a symbol on top of the stack
  - This definition allows nondeterminism since δ can return a set

How Does a PDA Compute?

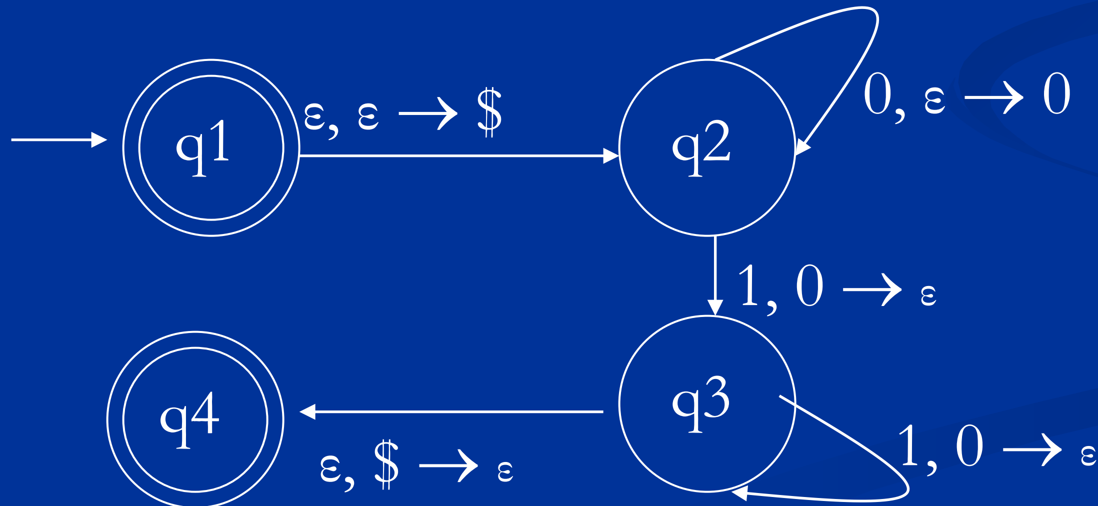
- The following 3 conditions must be satisfied for a string to be accepted:
    1. M must start in the start state with an empty stack
    2. M must move according to the transition function
    3. At the end of the input, M must be in an accept state
- To make it easy to test for an empty stack, a $ is initially pushed onto the stack
    - If you see a $ at the top of the stack, you know it is empty

Notation

- We write a,b $\rightarrow$ c to mean:
  - when the machine is reading an a from the input
    - it may replace the b on the top of the stack with c
  - Any of a, b, or c can be ε
    - If a is ε then can make stack change without reading an input symbol
    - If b is ε then no need to pop a symbol (just push c)
    - If c is ε then no new symbol is written (just pop b)

Example 1: a PDA for 0^n1^n

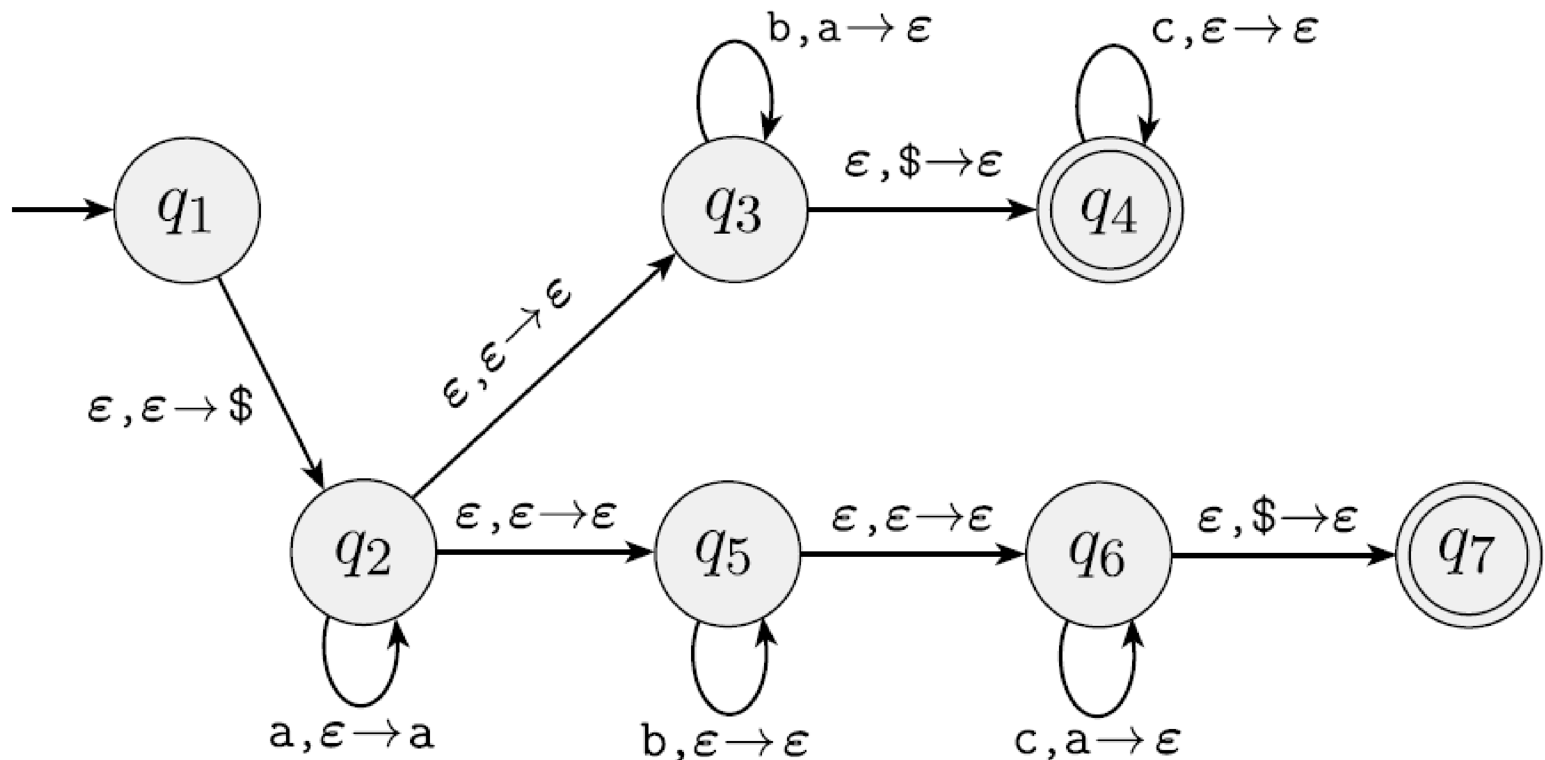
- Formally describe PDA that accepts $\{0^n1^n \mid n \geq 0\}$
  - Let M1 be $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where
    - $Q = \{q_1, q_2, q_3, q_4\}$ $\Sigma = \{0, 1\}$
    - $\Gamma = \{0, \$\}$ $F = \{q_1, q_4\}$



Example 2: PDA for $a^i b^j c^k$, i=j or i=k

- Come up with a PDA that recognizes the language $\{a^i b^j c^k \mid i, j, k \geq 0 \text{ and } i=j \text{ or } i=k\}$
  - Come up with an informal description, as we did initially for $0^n 1^n$
    - Can you do it without using non-determinism?
      - No
    - With non-determinism?
      - Easy, similar to $0^n 1^n$ except that guess whether to match a's with b's or c's. See Figure 2.17 page 114
    - Since NPDA $\neq$ PDA, create a PDA if possible

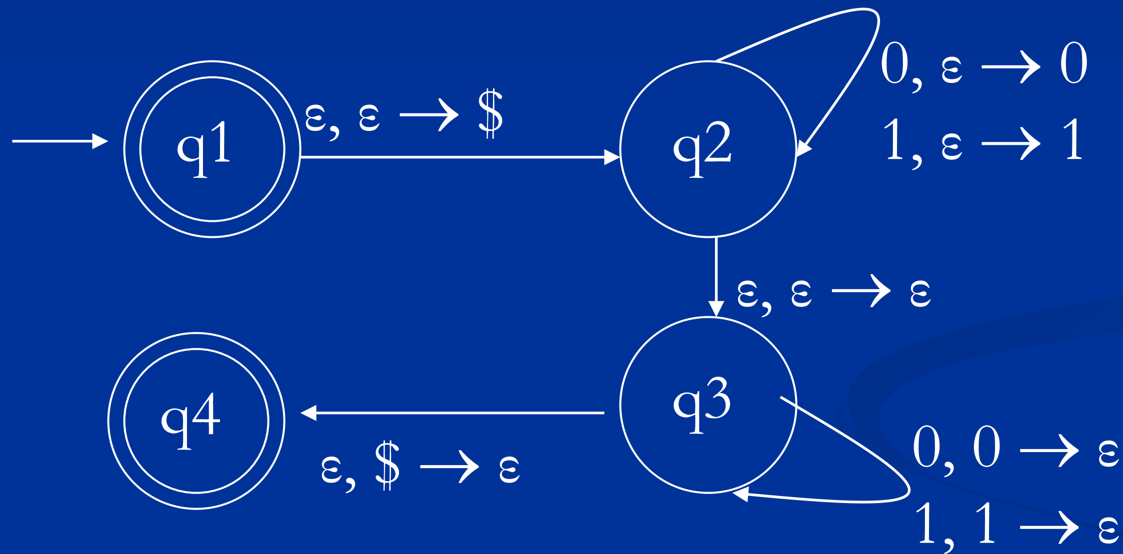
Example 2: PDA for $a^ib^jc^k$, i=j or i=k



Example 3: PDA for $\{ww^R\}$

- Come up with a PDA for the following language: $\{ww^R \mid w \in \{0,1\}^* \}$
  - Recall that w^R is the reverse of w so this is the language of palindromes
  - Can you informally describe the PDA? Can you come up with a deterministic one?
    - No
  - Can you come up with a non-deterministic one?
    - Yes, push symbols that are read onto the stack and at some point nondeterministically guess that you are in the middle of the string and then pop off stack value as they match the input (if no match, then reject)

Diagram of PDA for $\{ww^R\}$



Equivalence with CFGs

- Theorem: A language is context free if and only if some pushdown automaton recognizes it
  - Lemma: if a language L is context free then some PDA recognizes it (we won't bother doing other direction)
  - Proof idea: We show how to take a CFG that generates L and convert it into an equivalent PDA P
    - Thus P accepts a string only if the CFG can derive it
    - Each main step of the PDA involves an application of one rule in the CFG
    - The stack contains the intermediate strings generated by the CFG
    - Since the CFG may have a choice of rules to apply, the PDA must use its non-determinism
    - One issue: since the PDA can only access the top of the stack, any terminal symbols pushed onto the top of the stack must be checked against the input string immediately.
      - If the terminal symbol matches the next input character, then advance input string
      - If the terminal symbol does not match, then terminate that path

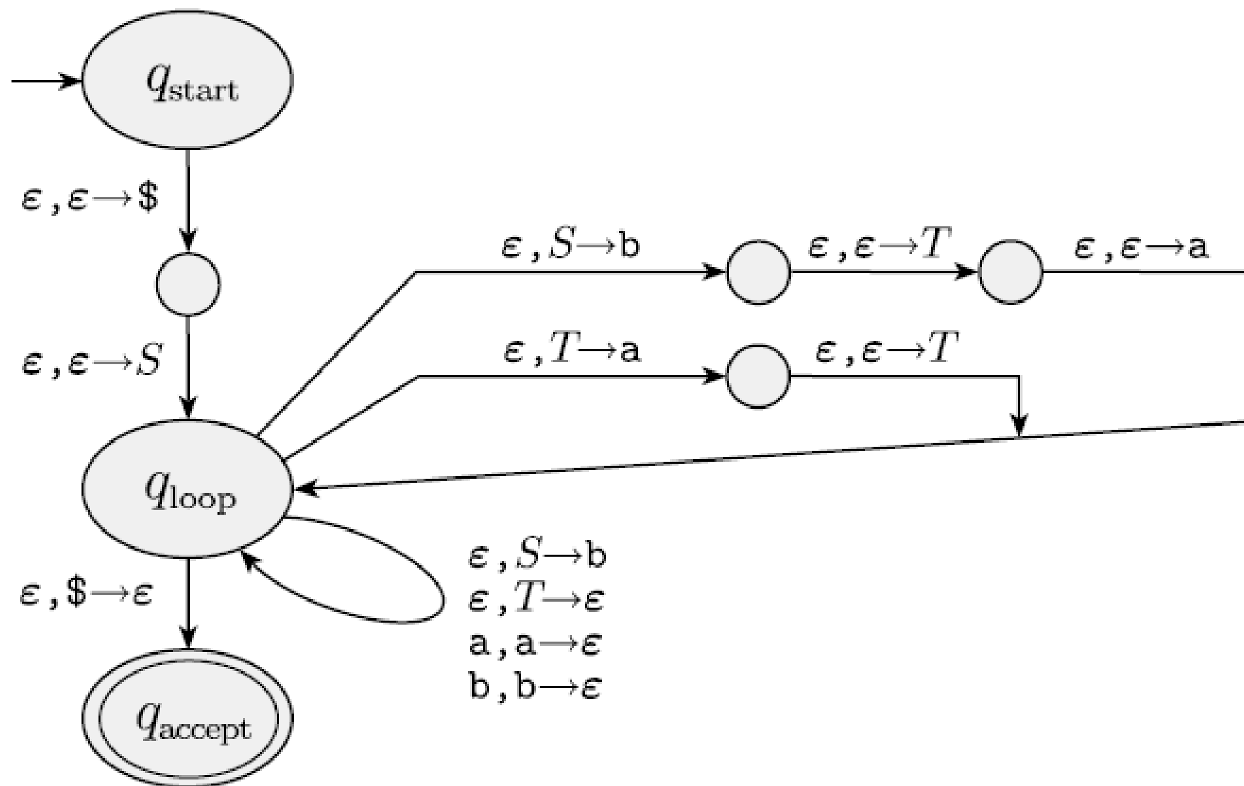
Informal Description of P

- Place marker symbol $ and the start variable on the stack
- Repeat forever
  - If the top of the stack is a variable A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule
  - If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject this branch.
  - If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

Example 2.25: construct a PDA P1 from a CFG G

$$S \to \mathtt{a}T\mathtt{b} \mid \mathtt{b}$$
$$T \to T\mathtt{a} \mid \varepsilon$$

The transition function is shown in the following diagram.



Example

- Note that the top path in q_{loop} branches to the right and replaces S with aTb
  - It first pushes b, then T, then a (a is then at top of stack)
- Note the path below that replaces T with Ta
  - It replaces T with a then pops T on top of that
- Your task:
  - Show how this PDA accepts the string aab, which has the following derivation:
    - $S \rightarrow aTb \rightarrow aTab \rightarrow aab$

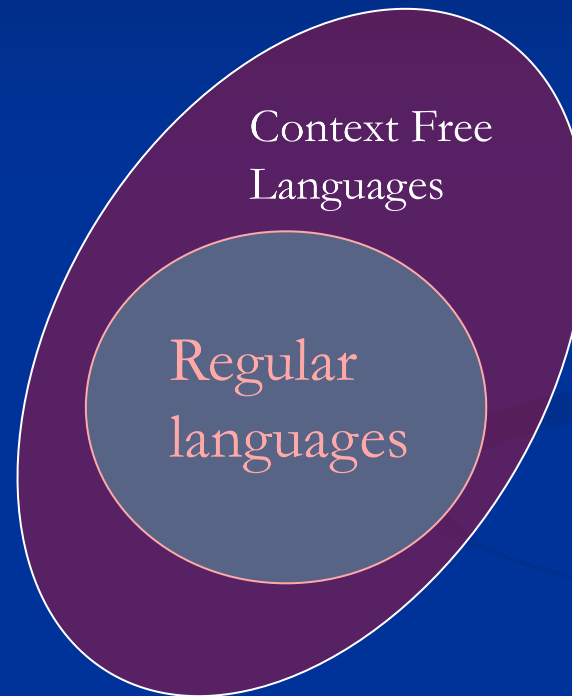
Example continued

- In the following, the left is the top of stack
  - We start with S$
    - We take the top branch to the right and we get the following as we go thru each state:
      - S$ → b$ → Tb$ → aTb$
    - We read a and use rule a,a → ε to pop it to get Tb$
    - We next take the 2nd branch going to right:
      - Tb$ → ab$ → Tab$
    - We next use rule ε,T → ε to pop T to get ab$
    - Then we pop a then pop b at which point we have $
    - Everything read so accept

Relationship of Regular Languages & CFLs

- We know that CFGs define CFLs
- We now know that a PDA recognizes the same class of languages and hence recognizes CFLs
- We know that every PDA is a FA that just ignores the stack
- Thus PDAs recognize regular languages
- Thus the class of CFLs contains regular languages
- But since we know that a FA is not as powerful as a PDA (e.g., 0^n1^n) we can say more
  - CFLs and regular languages are not equivalent

Relationship between CFLs and Regular Languages



Section 2.3

Non-Context Free Languages

Non Context Free Languages

- Just like there are languages that are not regular, there are languages that are not context free
  - This means that they cannot be generated by a CFG
  - This means that they cannot be generated by a PDA
- Just your luck! There is also a pumping lemma to prove that a language is not context free!

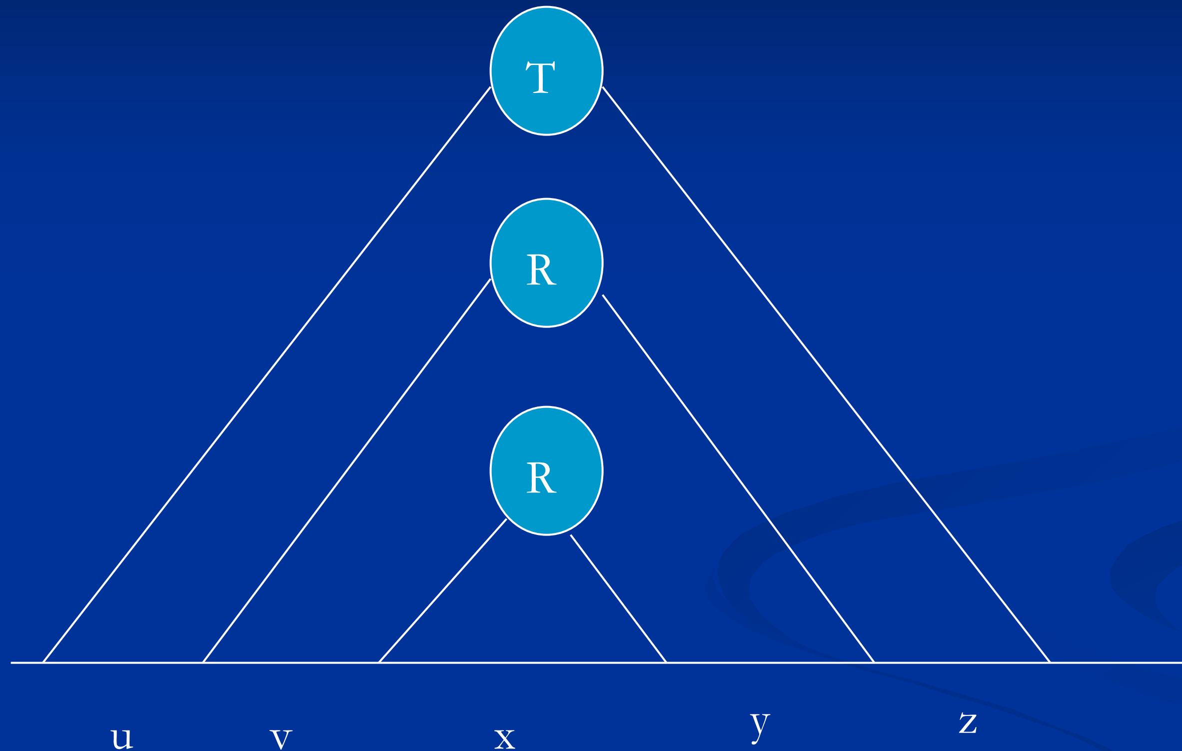
Pumping Lemma for CFGs

- If A is a context-free language then there is a pumping length p where, if s is any string in A with length $\geq$ p, then s may be divided into five pieces x = uvxyz satisfying 3 conditions:
  1. For each $i \geq 0$, $uv^ixy^iz \in A$
  2. $|vy| > 0$, and
  3. $|vxy| \leq p$

Proof Idea

- For regular languages we applied the pigeonhole principle to the number of states to show that a state had to be repeated.
  - Here we apply the same principle to the number of variables in the CFG to show that some variable will need to be repeated given a sufficiently long string
    - We will call this variable R and assume it can derive X
  - I don’t find the diagram on the next slide as obvious as the various texts suggest. However, if you first put the CFG into a specific form, then it is a bit clearer.
    - Mainly just be sure you can apply the pumping lemma

Proof Idea Continued



$T \rightarrow uRz \quad R \rightarrow x \quad R \rightarrow vRy$

Since we can keep applying rule $R \rightarrow vRy$, we can derive uv^ixy^iz which therefore must also belong to the languages

Example I

- Use the pumping lemma to show that the language $B = \{a^n b^n c^n \mid n \geq 0\}$ (Ex 2.36)
  - What string should we pick?
  - Select the string $s = a^p b^p c^p$. $S \in B$ and $|s| > p$
  - Condition 2 says that v and y cannot both be empty
  - Using the books reasoning we can break things into two cases (note that $|vxy| \leq p$). What should they be or how would you proceed?
    - v and y each only contain one type of symbol (one symbol is left out)
      - uv^2xy^2z cannot contain equal number of a's, b's and c's
    - Either v or y contains more than one type of symbol
      - Pumping will violate the separation of a's, b's and c's
        - We used this reasoning before for regular languages
  - Using my reasoning
    - Since $|vxy| \leq p$ v and y contain at most 2 symbols and hence at least one is left out when pump up (technically we should say at least one symbols is in v or y so that pumping break the equality)

Example II

- Let $D = \{ww \mid w \in \{0,1\}^*\}$ (Ex 2.38)
  - What string should we pick?
  - A possible choice is to choose $s = \{0^p10^p1\}$
    - But this can be pumped– try generating uvxyz so it can be pumped
    - Hint: we need to straddle the middle for this to work
    - Solution: $u=0^{p-1}$, $v=0$, $x=1$, $y=0$, $z=0^{p-1}1$
    - Check it. Does $uv^2xy^2z \in D$? Does $uwz \in D$?
  - Choose $s = \{0^p1^p0^p1^p\}$
    - First note that vxy must straddle midpoint. Otherwise pumping makes 1st half ≠ 2nd half
      - Book says another way: if vxy on left, pumping it moves a 1 into second half and if on right, moves 0 into first half
        - since $|vxy| < p$, it is all 1's in left case and all 0's in right case
    - If does straddle midpoint, if pump down, then not of form ww since neither 1's or 0's match in each half

Summary of Pumping Lemma for CFGs

- Just remember the basics and the conditions
  - You should memorize them for both pumping lemmas, but think about what they mean
  - I may give you the pumping lemma definitions on the exam, but perhaps not. You should remember both pumping lemmas and the conditions
    - They should not be too hard to remember