CSCI 2670, Fall 2012

Introduction to Theory of Computing

Department of Computer Science
University of Georgia
Athens, GA 30602

Instructor: Liming Cai
www.cs.uga.edu/∼cai

Lecture Note 3

Context-Free Languages

Chapter 2. Context-free Languages

- We know there are languages that are NOT regular;
- Are all these non-regular languages of the 'same difficulty'?
- How can we define these non-regular languages rigorously?
- Are there more difficult languages than $\{0^n1^n : n \geq 0\}$?

- We first investigate a class of languages
  called 'context-free languages'
- Context-free languages have extensive applications in
  programming language and compiler designs.
- CFL can be defined in a rigorous way, similar to regular languages

  formal system (Push-down automata) to recognize
  formal system (context-free grammars) to define
  pumping lemma also
- context-free grammars ALSO allow us to see how to define
  regular languages syntactically.

2.1 Context-free grammars

We begin by examining finite automata that recognize regular languages, as **an introduction** to grammar systems.

Example 1: A DFA without a circular path for $L = \{01, 1, 00\}$

- draw a DFA with start state S, accept states B, C, and D, and another state A.

- convert the DFA to 'grammar rules':

  $S \to 0A$, $A \to 0C$, $A \to 1D$, $S \to 1B$.

- generating strings of the language L (called *derivation*) by applications of rules: LHS letter is replaced by RHS letters

- we can remove the symbols representing accepting states.

- derivation path:
  the sequence of rule applications to get a string,

  deriving: string 00: $S \Rightarrow 0A \Rightarrow 00$

Example 2: DFA with a loop: $L = \{01^n : n \geq 1\}$

- draw a DFA of start state S, accepting state B and another state A
- convert the DFA to 'grammar rules':

  $S \to 0A$,   but $A \to 1B$, $B \to 1B$?

  Two solutions:

  (a) 'combine' A and B: $A \to 1$, $A \to 1A$

  (b) add 'ϵ-rule': $B \to 1B$, $B \to \epsilon$

- deriving string 0111:

  $S \Rightarrow 0A \Rightarrow 01B \Rightarrow 011B \Rightarrow 0111B \Rightarrow 0111\epsilon = 0111$

Example-3 A little more complicated regular language: 011^*0^*

- draw a DFA, but how?

- convert it to 'grammar rules':
  $S \to 0A$, $A \to 1B$, $B \to 1B$, $B \to \epsilon$,
  $B \to C$, $C \to \epsilon$, $C \to 0C$.

- deriving string 011, 0110, 011000?

we can consolidate rules to simply the grammar.

Summarize the **regular grammar** rules:

$Y \to \gamma$

- a single substitutable symbol, called *nonterminal*, as LHS

- γ contains at most 2 symbols as RHS:

  (1) aX, $a \in \Sigma$, called *terminal*, X is nonterminal, or

  (2) a, $a \in \Sigma$, or

  (3) X, a nonterminal, or

  (4) ϵ, empty string.

Now we consider to loosen constraints to the regular grammar rules:

(1) allow more than one terminals in the rules

$S \to 01A$, $A \to \epsilon$, $A \to 1A$.

- does not seem to increase the power, but

(2) allow more than terminals on both sides of a nonterminal

$S \to 0A1$, $A \to S$, $S \to \epsilon$

- what language does it generates?

- it contains $\epsilon, 01, 0011, 000111$, etc..

(3) only allow terminals one side nonterminal X at a time

e.g., $S \to 0A$, $A \to S1$, $S \to \epsilon$

what language does it generates?

- it contains $\epsilon, 01, 0011, 000111$, etc..

called a **linear grammar**

(4) How about the following language whose strings are paired parentheses, like $((()())())$

- (2) and (3) only allow to generate $((()))$ type of strings
- allow more than one nonterminals in RHS

$S \to (S)$, $S \to \epsilon$, $S \to AB$, $A \to S$, $B \to S$. (simplify it!)

deriving $((()())())$

$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow ((S)S) \Rightarrow ((SS)S)$
$\Rightarrow (((S)S)S) \Rightarrow ((()S)S) \Rightarrow ((()(S))S) \Rightarrow ((()())S)$
$\Rightarrow ((()())(S)) \Rightarrow ((()())())$

How about $L = \{w : w \text{ contains the same number of 1s and 0s}\}$?

Derivation tree (parsing tree):

Drawing a tree for a derivation process for a generated string: $((()())())$

- the start symbol is the root;
- LHS nonterminal symbol is a parent
- symbols in the RHS are children

chaining all the leaves results in the generated/derived string.

Formal Definition of a context-free grammar

Definition 2.2

A *context-free grammar* is a 4-tuple (V, Σ, R, S), where

1. V is a finite set called variables;
2. Σ is a finite set, disjoint from V, called terminals;
3. R is a finite set of rules, each of the format $X \to \gamma$, where $X \in V$, $\gamma \in (\Sigma \cup V)^*$; and
4. $S \in V$ is the start variable.

Formal definition of *dervation*:

Let $u, vw \in (\Sigma \cup V)^*$, and $A \rightarrow w$ be a rule. Then we say string uAv *yields* string uwv, written as $uAV \Rightarrow uwv$.

Let $\alpha, \beta \in (\Sigma \cup V)^*$. We say α *derives* β, written as $\alpha \Rightarrow^* \beta$, if

(1) $\alpha = \beta$, or

(2) $\alpha \Rightarrow \alpha_1$, and $\alpha_1 \Rightarrow \alpha_k \Rightarrow \beta$,
for some $\alpha_1, \alpha_2, \ldots, \alpha_k \in (\Sigma \cup V)^*$, for some $k \geq 0$.

Note condition (2) can be written as:

(2) $\alpha \Rightarrow \gamma$, and $\gamma \Rightarrow^* \beta$, for some $\gamma \in (\Sigma \cup V)^*$

Now back to

Formal Definition of a context-free grammar

Definition 2.2

A *context-free grammar* is a 4-tuple $G = (V, \Sigma, R, S)$, where

1. V is a finite set called variables;
2. Σ is a finite set, disjoint from V, called terminals;
3. R is a finite set of rules, each of the format
   $X \to \gamma$, where $X \in V$, $\gamma \in (\Sigma \cup V)^*$; and
4. $S \in V$ is the start variable.

Define *the language of the grammar* G to be

$$L(G) = \{w : w \in \Sigma^*,\ S \Rightarrow^* w\}$$

Example 2.3, page 103

$G_3 = (\{S\}, \{a, b\}, R, S)$, where the set of rules, R, is

$S \to aSb \,|\, SS \,|\, \epsilon$

where | represents 'or' for multiple rules sharing the same LHS.

derivation, parsing tree for string $abaababa$

Example 2.4 the language of "arithmetic expressions"

$G_4 = (V, \Sigma, R, S\}$ where

$V = \{\langle exp\rangle, \langle term\rangle, \langle factor\rangle\}$, $\Sigma = \{a, +, \times\}$, $S = \langle exp\rangle$

R is:

$\langle exp\rangle \to \langle exp\rangle + \langle term\rangle \mid \langle term\rangle$

$\langle term\rangle \to \langle term\rangle \times \langle factor\rangle \mid \langle factor\rangle$

$\langle factor\rangle \to a$

add parentheses (and) to the expressions:

$\langle factor\rangle \to (\langle exp\rangle) \mid a$

parsing trees for $a + a \times a$ and $(a + a) \times a$.

Designing context-free grammars

1. familiar with rules

   - simple rules
   - recursive rules - almost always needed
   - ϵ-rule - almost always needed for recursive patterns
   - bifurcation rules - other than purely nested structure

2. simplification of rules

   - make sure the grammar is correct first - removable rules - how simple is simple?

Ambiguity

A grammar is *ambiguous* if it can derive the same string in more than one way.

Examples:

$$E \to E + E$$
$$E \to E \times E$$
$$E \to a$$

Formally, a derivation of a string w in a grammar G is a
leftmost derivation
if at every step the leftmost remaining variable is the one replaced.

Definition 2.7 A string w is derived **ambiguously** in grammar G if it has two or more different leftmost derivations.

A grammar G is *ambiguous* if it generates some string ambiguously.

Note: Sometimes for an ambiguous grammar, we can find a non-ambiguous grammar that generated the same language.

A language is **inherently ambiguous** if it can only be generated by ambiguous grammars.

Chomsky Normal Form

Motivation: We simplify context-free grammar rules, for the purpose of designing simpler algorithms to recognize the languages generated by CF grammars.

Definition 2.8 A context-free grammar is in **Chomsky normal form** if every rule is of the form

$A \to BC$, or

$A \to a$

where a is a terminal and A, B, and C are variables - except that B and C may not be the start variable. In addition, we permit the rule $S \to \epsilon$ for start variable only.

Theorem 2.9 Any CFL is generated by some CFG in Chomsky normal form.

(1) That is, for every CFG, there is a CFG in Chomsky normal form that generates the same language.

(2) The proof of the theorem explicits transforms a CFG into Chomsky normal form.

- add a new start variable
- remove all ϵ-rules: $A \to \epsilon$
- eliminate all unit rules: $A \to B$
- path up the grammar to generate the same language
- convert remaining rules into the desired form.

Work on examples – before do a formal proof for the theorem!

Example 2.10

$S \to ASA \quad S \to aB$
$A \to B \quad A \to S$
$B \to b \quad B \to \epsilon$

Steps:

(1) add $S_0 \to S$
(2) remove $B \to \epsilon$ (but also create $A \to \epsilon$) and remove $A \to \epsilon$
(3) remove $S \to S$, $S_0 \to S$
(4) remove $A \to B$, $A \to S$
(5) add new variables
- replacing a terminal
- replacing two variables

Proof:

Show that all steps of transforming a CFG to a Chomsky normal form does not change the language it accepts.

(1) add a new start variable S_0 and rule $S_0 \rightarrow S$
where S is the old start variable.

(2) remove ϵ rules $A \rightarrow \epsilon$

for every rule $B \rightarrow \alpha A \beta$ and every occurrence of A,
add new rule $B \rightarrow \alpha\beta$, where $\alpha, \beta \in (V \cup \Sigma)^*$

note: removing $A \rightarrow \epsilon$ may create new ϵ rules for B.
so repeating the process when needed.

(3) remove unit rules $A \to B$

for every rule $B \to \alpha$, add a new rule $A \to \alpha$.

note: this may create unit rules for A as well,
so repeat the process when needed.

(4) convert to the proper form (patching up the rules)

for every rule $A \to x_1 x_2 \dots x_k$

(a) if $x_i \in \Sigma$, add rule $X_i \to x_i$
(b) if $x_i \in V$, then $X_i = x_i$ (keep the variable).

(c) add new rules:

$A \to X_1 A_1$
$A_1 \to X_2 A_2$
$A_2 \to X_3 A_3$
$\cdots$
$A_{k-2} \to X_{k-1} X_k$

Example for step (4): $A \to aBcD$,

in the general format $A \to x_1x_2x_3x_4$,
where $x_1 = a$, $x_2 = B = X_2$, $x_3 = c$, $x_4 = D = X_4$.

so add rules $X_1 \to a$, $X_3 \to c$

Then add rules

$A \to X_1A_1$
$A_1 \to BA_2$
$A_2 \to X_3D$

2.2 Pushdown Automata PDA

finite state machines equipped with a stack

every time a symbol is read,

- there is a state transition
- there is an operation in the stack

Recall: operations on a stack S:

- $\textsc{Push}(S, a)$
- $\textsc{Pop}(S)$
- $\textsc{Top}(S)$

combined rule: old top-of-stack $\longrightarrow$ new top-of-stack

- $\textsc{Push}(S, a)$: $\epsilon \longrightarrow a$
- $\textsc{Pop}(S)$: $a \longrightarrow \epsilon$
- $\textsc{Top}(S)$: $a \longrightarrow a$

Formal definition of a PDA

working of a PDA: given

a input symbol, current state, current stack top content

state change, stack top content change

So we need

Σ, Γ, Q, but to allow nondeterminism,

use $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$, $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$

domain of transition function: $Q \times \Sigma_\epsilon \times \Gamma_\epsilon$

range of transition function: $\mathcal{P}(Q \times \Gamma_\epsilon)$

Definition 2.13 page 111

A *pushdown automaton* is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ and F are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\epsilon)$,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

See some examples before formal definition of computation with PDAs.

The following PDA recognizes language $\{0^n 1^n | n \geq 0\}$

$$(Q, \Sigma, \Gamma, \delta, q_0, F)$$

where $Q = \{q_1, q_2, q_3, q_4\}$,
$\Sigma = \{0, 1\}$, $\Gamma = \{0, \$\}$, $F = \{q_1, q_4\}$

and δ

- $\delta(q_1, \epsilon, \epsilon) = \{(q_2, \$)\}$, $\delta(q_2, 0, \epsilon) = \{(q_2, 0)\}$,
- $\delta(q_2, 1, 0) = \{(q_3, \epsilon)\}$, $\delta(q_3, 1, 0) = \{(q_3, \epsilon)\}$,
- $\delta(q_3, \epsilon, \$) = \{(q_4, \epsilon)\}$, and
- mapping to empty set for all others domain values.

Follow the PDA on some string examples

State diagrams for PDAs

- following FA diagrams
- on the transition edge, stack operation as well as the symbol

$a, b \longrightarrow c$:

read input symbol a, stack top b, update stack with c

$a, \epsilon \longrightarrow c$ means push

$a, b \longrightarrow \epsilon$ means pop

$a, b \longrightarrow c$ means replace

Figure 2.15 (page 113)

try to relate this to a DFA recognizing regular language 0^*1^*.

Issues about testing empty stack and testing the end of input

- we can put special symbol $ to the stack in the beginning and once we see it again, it is the end of stack
- a PDA cannot test the end of the input string, accepting a string when at an accept state and the end of string (as defined!)

We need a formal definition of accepting a language by a PDA

A PDA $(Q, \Sigma, \Gamma, \delta, q_0, F)$ computes as follows.

It *accepts* string w if

(a) w can be written as $w = w_1 w_2 \ldots w_m$,
where $w_i \in \Sigma_\epsilon$, $i = 1, 2, \ldots, m$,

(b) there is a sequence of states $r_0, r_1, \ldots, r_m$, and

(c) there are strings $s_0, s_1, \ldots, s_m \in \Gamma^*$

such that

1. $r_0 = q_0$ and $s_0 = \epsilon$,
2. For $i = 0, 1, \ldots, m-1$, we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$,
   where $s_i = at$, $s_{i+1} = bt$ for some $a, b \in \Gamma_\epsilon$ and $t \in \Gamma^*$
3. $r_m \in F$.

Again, use of nondeterminism

- nondeterministic computation model is hypothetical
- endowed with the power to guess ‘correctly’
- particularly useful in computation with many options that include the correct one
- nondeterminism is able to ‘guess’ and ’pursue’ the correct judgement

e.g., NFA to recognize strings that contain pattern ‘11’.

Example 2.16 page 113

PDA to recognize language

$$\{a^i b^j c^k \mid i, j, k \geq 0, \text{ and } i = j \text{ or } i = k\}$$

- for the input string $a...ab...bc...c$, there are two possibilities

  (1) $a^i b^i c^k$, or
  (2) $a^i b^k c^i$

- computation *nondeterministically* choose (1) or (2)

- but PDA has to accommodate both scenarios and
  let the computation choose

- Figure 2.17 page 114

Example 2.18 page 114

PDA to recognize $\{ww^R \mid w \in \{0,1\}^*\}$, w^R is w written backward.

Idea:

- pushing symbols in w to the stack
- popping them out to match symbols in w^R
- how do we know when to stop pushing (and start popping)?
- use nondeterminism!

At each encounter with a symbol in the input,

the PDA nondeterministically choose the following two options

(a) it is right past the midpoint: start popping the stack

(b) it is still before the midpoint: keep pushing to the stack

which implies a branching in the PDA

Figure 2.19 page 114

Equivalence with Context-Free Grammars

Theorem 2.20 A language is context-free if and only if some PDA recognizes it.

Lemma 2.21 Every CFL is recognized by a PDA.

Lemma 2.27 Every language recognized by PDA is CFL.

Lemma 2.21 Every CFL is recognized by a PDA.

proof idea;

- assume a CFG for the given language L
- following the production rules simulate string derivations
- use nondeterminism for the multiple options of rules
- example: $L = \{0^k 1^k \mid k \geq 0\}$, with the corresponding grammar of rules:

  $S \rightarrow \epsilon \mid 0S1$

  a PDA will use production rules nondeterministically to derive a string that matches the query string

Question: without knowing L explicitly, how do you recognize the language defined by a CFG **?**

A PDA simulates derivation of s based on grammar rules.

For example $s = 0011$

input	**derivation**	**stack**	**rule selected to use**
0011	**S**	**S**	$S \to 0S1$
$\underline{0}011$	$\underline{0}S1$	$\underline{0}S1$	
$\underline{0}011$	$\underline{0}\mathbf{S}1$	$\mathbf{S}1$	$S \to 0S1$
$\underline{00}11$	$\underline{00}S11$	$\underline{0}S11$	
$\underline{00}11$	$\underline{00}\mathbf{S}11$	$\mathbf{S}11$	$S \to \epsilon$
$\underline{001}1$	$\underline{001}1$	$\underline{1}1$	
$\underline{001}1$	$\underline{001}1$	$\underline{1}$	
$\underline{0011}$	$\underline{0011}$	empty	

Matches are underscored; bold nonterminal to be expanded.

A more systematic idea:

$S \to 0S1 \mid 1S0 \mid SS \mid \epsilon$

(yes, only one nonterminal S but it has 4 alternative rules!)

e.g., $s = 0110$, the following are steps to recognize s

input	**derivation**	**stack**	**rule selected to use**
0110	$\mathbf{S}$	$\mathbf{S}$	$S \to SS$
0110	$\mathbf{S}S$	$\mathbf{S}S$	$S \to 0S1$
$\underline{0}110$	$\underline{0}S1S$	$\underline{0}S1S$	
$\underline{0}110$	$\underline{0}\mathbf{S}1S$	$\mathbf{S}1S$	$S \to \epsilon$
$\underline{01}10$	$\underline{01}S$	$\underline{1}S$	
$\underline{01}10$	$\underline{01}\mathbf{S}$	$\mathbf{S}$	$S \to 1S0$
$\underline{011}0$	$\underline{011}S0$	$\underline{1}S0$	
$\underline{011}0$	$\underline{011}\mathbf{S}0$	$\mathbf{S}0$	$S \to \epsilon$
$\underline{0110}$	$\underline{0110}$	$\underline{0}$	
$\underline{0110}$	$\underline{0110}$	empty	

A PDA that can accomplish the work in the previous table needs to:

1. if the stack top is a variable, nondeterministically select a rule to apply, and replace the stack top (LHS of the selected rule) with RHS
2. if the stack top is a terminal, match the current input symbol pop the stack
3. if does not match, reject
4. if match all input symbols and stack is empty, accept
5. if stack is empty but not finish all symbols and not at the start state, reject

See if we can construct a PDA based on the grammar!

- $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, S, \$\}$

- need a q_0, but Q and F to be determined

transition function δ is defined as

$\delta(q_0, \epsilon, \epsilon) = \{(q_1, S\$)\}$ **what should S be in general?**
$\delta(q_1, \epsilon, S) = \{(q_1, 0S1), (q_1, 1S0), (q_1, SS), (q_1, \epsilon)\}$
$\delta(q_1, 0, 0) = \{(q_1, \epsilon)\}$ $\delta(q_1, 1, 1) = \{(q_1, \epsilon)\}$
$\delta(q_1, \epsilon, \$) = \{(q_2, \epsilon\}$

How to push multiple symbols?

$\delta(q_0, \epsilon, \epsilon) = \{(q_1, S\$)\}$ can be accomplished with

$$\delta(q_0, \epsilon, \epsilon) = \{(q_1', \$)\}, \quad \delta(q_1', \epsilon, \epsilon) = \{(q_1, S)\}$$

Also $\delta(q_1, \epsilon, S) = \{(q_1, 0S1), (q_1, 1S0), (q_1, SS), (q_1, \epsilon)\}$ can be accomplished with

$$\delta(q_1, \epsilon, S) = \{(q_3, 1), (q_4, 0), (q_5, S), (q_1, \epsilon)\}$$

$$\delta(q_3, \epsilon, \epsilon) = \{(q_3', S)\}, \quad \delta(q_3', \epsilon, \epsilon) = \{(q_1, 0)\}$$

$$\delta(q_4, \epsilon, \epsilon) = \{(q_4', S)\}, \quad \delta(q_4', \epsilon, \epsilon) = \{(q_1, 1)\}$$

$$\delta(q_5, \epsilon, \epsilon) = \{(q_1, S)\}$$

PDA diagram to illustrate!

Procedure to construct a PDA from a CFG (page 116)

1. Push the special symbl $ and start nonterminal in the stack
2. Do the following steps

   (1). If the top of stack is a nonterminal A,

   - nondeterministically select one of its rules, and
   - substitute A with the RHS, goto step 2

   (2). If the top of stack is a terminal a,

   - read the next symbol from the input and compare to a,
   - if match, goto step 2; otherwise, reject and stop.

   (3). If the top of stack is $, enter the accept state.

   - if all input has been read, accept and stop, otherwise goto step 2.

Proof (outline, details page 116-117)

1. The PDA has Σ the same as the alphabet as the grammar.

2. Γ consists of both terminals and nonterminals of the grammar.

3. $Q = \{q_{start}, q_{loop}, q_{accept}\} \cup E$, where E contains those states needed by pushing mutliple symbols into stacks.

4. q_{accept} is the only accept state.

5. $\delta(q_{start}, \epsilon, \epsilon) = \{(q_{loop}, S\$)\}$
   $\delta(q_{loop}, \epsilon, A) = \{(q_{loop}, w) \mid A \to w \text{ is a grammar rule}\}$
   $\delta(q_{loop}, a, a) = \{(q_{loop}, \epsilon)\}$
   $\delta(q_{loop}, \epsilon, \$) = \{q_{accept}, \epsilon)\}$

Figure 2.24: schematic diagram for the constructed PDA (page 118)

Example 2.25

Construct a PDA for the language described by the following grammar:

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

What do the strings look like BTW?

We following proof of Lemma 2.21.

Lemma 2.27 Every language recognized by PDA is CFL.

proof idea:

To construct a CFG for each PDA.

(Recall how we did to prove every language recognized by DFA has a regular expression)

For every pair of states p and q,

define a nonterminal $A_{p,q}$, and rules

- such that A_{pq} generates all strings taking the PDA from p to q
- leaving the stack at q the same condition as it was at p
- (**the same as from empty stack to empty stack**)

The PDA moves from p to q by pushing and popping stack

(1) either push some x at the beginning and pop x at the end
(2) or push x at the beginning and pop it out in the middle

For (1), we create rule $A_{pq} \rightarrow aA_{rs}b$, where
r is a state following p and s preceeds q, and
a is the first symbol on the string, and b is the last

For (2), we create rule $A_{pq} \rightarrow A_{pr}A_{rq}$, where
r is the state with the stack returning to the same status as state p.

Example, recall the PDA we construct for language $\{0^n 1^n \mid n \geq 0\}$

$Q = \{q_1, q_2, q_3, q_4\},\ \Sigma = \{0, 1\},\ \Gamma = \{0, \$\},\ F = \{q_4\}$

- $\delta(q_1, \epsilon, \epsilon) = \{(q_2, \$)\}$, $\delta(q_2, 0, \epsilon) = \{(q_2, 0)\}$,
- $\delta(q_2, \epsilon, \epsilon) = \{(q_3, \epsilon)\}$, $\delta(q_3, 1, 0) = \{(q_3, \epsilon)\}$,
- $\delta(q_3, \epsilon, \$) = \{(q_4, \epsilon)\}$, and

Create $A_{q_2 q_3} \rightarrow \epsilon$, $\quad A_{q_2 q_3} \rightarrow 0 A_{q_2 q_3} 1$

also $A_{q_1 q_4} \rightarrow \epsilon A_{q_2 q_3} \epsilon$.

Convert these rule into (simplified):

$S \rightarrow A$
$A \rightarrow \epsilon | 0A1$

or more simply:

$S \rightarrow \epsilon | 0S1$

Another example: Figure 2.17 (Page 114), a PDA to recognize language $\{a^i b^j c^k \mid i, j, k \geq 0, \text{ and } i = j \text{ or } i = k\}$

Create nonterminals and production rules:

$A_{q_2,q_3} \to \epsilon$	$X \to \epsilon$
$A_{q_2,q_3} \to aA_{q_2,q_3}b$	$X \to aXb$
$A_{q_4,q_4} \to \epsilon$	$Y \to \epsilon$
$A_{q_4,q_4} \to cA_{q_4,q_4}$	$Y \to cY$
$A_{q_1,q_4} \to A_{q_2,q_3}A_{q_4,q_4}$	$S_1 \to XY$
$A_{q_5,q_5} \to \epsilon$	$W \to \epsilon$
$A_{q_5,q_5} \to bA_{q_5,q_5}$	$W \to bW$
$A_{q_2,q_6} \to aA_{q_2,q_6}c$	$U \to aUc$
$A_{q_2,q_6} \to A_{q_5,q_5}$	$U \to W$
$A_{q_1,q_7} \to A_{q_2,q_6}$	$S_2 \to U$

Proof: Assume that the PDA has the following features:

(1) It has a single accepting state, q_{accept}.

(2) It empties its stack before accepting.

(3) Each transition either pushes a symbol or pop a symbol, but not both at the same time.

Assume PDA $(Q, \Sigma, \Gamma, \delta, q_0, \{q_{accept}\})$ and construct a CFG grammar , such that

- variable set $V = \{A_{p,q} \mid p, q \in Q\}$.
- start variable $A_{q_0, q_{accept}}$.
- for each $p, q, r, s \in Q$, $t \in \Gamma$ and $a, b \in \Sigma_\epsilon$,
  if $\delta(p, a, \epsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ϵ)
  create rule $A_{p,q} \rightarrow aA_{r,s}b$
- for each $p, q, r \in Q$, create rule $A_{p,q} \rightarrow A_{p,r}A_{r,q}$.
- for each $q \in Q$, create rule $A_{q,q} \rightarrow \epsilon$.

Claim 2.30

$A_{p,q}$ generates string x, then x takes the PDA from state p with empty stack to state q with empty stack.

Claim 2.31

If string x takes the PDA from state p with empty stack to state q with empty stack, then $A_{p,q}$ generates x

Claim 2.30

$A_{p,q}$ generates string x, then x takes the PDA from state p with empty stack to state q with empty stack.

Proof: Consider $A_{p,q} \Rightarrow^* x$ and induction on the derivation length m.

Assume that the claim is true for derivation $m \leq k$.

Show for derivation of length $k+1$, all three cases give the claimed result.

Page 121.

Claim 2.31

If string x takes the PDA from state p with empty stack to state q with empty stack, then $A_{p,q}$ generates x

Proof: by induction on the number of steps that the PDA goes from state p with empty stack to state q with empty stack on input x

Page 122.

Non-Context-Free Languages

Is language $\{a^n b^n c^n \mid n \geq 0\}$ context-free?

try to use a single stack and finite number of memory cells to recognize such strings.

It is not context-free!

But how to prove?

Answer: Pumping lemma.

Recall the *Pumping Lemma* for regular language is based on the observations

- there are many strings longer than 'pump length'.
- the accepting path for such a long string goes *through the same state twice*.
- the substring on the circular subpath can be repeated/pumped so that **some longer strings belong to the language**.

To use the Pumping Lemma to prove a language is not regular,

- one needs to show those longer (pumped) strings do not maintain some property critical to the language, i.e., **they do not belong to the language**.

Circular paths are essential for a regular language
to contain long strings.

What is essential for a CFL to contain long strings?

Assume that s is a *very* long string in language A.

- Then s has a very tall derivation/parsing tree.
- There is a very long path from the root to some terminal in s.
- On this very long path, some nonterminal appear twice.
- That means, there is a derivation $R \Rightarrow^* vRy$.
- So the grammar allows pumping:

$$R \Rightarrow^* v^2Ry^2, \quad R \Rightarrow^* v^3Ry^3, \quad \ldots.$$

So if s is long enough, there exists a partition s into 5 parts:

$s = uvxyz$, in which v and y can be *simultaneously* pumped.

Theorem 2.34 Pumping Lemma for context-free grammar (page 123)

If A is a CFL, then there is a number p (the pumping length) such that, if $s \in A$, and $|s| \geq p$, then s can be written as $s = uvxyz$ satisfying conditions:

1. for each $i \geq 0$, $uv^ixy^iz \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.

Proof of Theorem 2.34

We already outlined the proof idea for why s can be pumped.

It remains to show

- what p is,
- proof for $|vy| > 0$ (what does it mean?)
- proof for $|vxy| \leq p$.

We want p to be **such** that

if $|s| \geq p$, there are multiple occurrences of some nonterminal R in a path from the root to a leaf in the parsing tree of s.

The biggest derivation tree, without repetition of nonterminals on paths, is a full b-ary tree.

If the grammar has $|V|$ nonterminals, the number of leaves is $b^{|V|}$.

So adding another level would make repetition of nonterminals on paths.

We set $p = b^{|V|+1}$

To prove $|vy| > 0$,

we want the parsing tree τ generating s to be the minimum, consisting of the smallest number of nodes.

If $|v| = |y| = 0$, there will be a small parsing tree generating s.

To ensure $|vxy| \leq p$,

we pick R in the bottom $|V| + 1$ nonterminals on the path.

Using the Pumping Lemma to prove that $L = \{a^n b^n c^n \mid n \geq 0\}$ is not context-free.

Assume L to be CF. Then there is a p, such that,

string $s = a^p b^p c^p \in L$ can be written as $s = uvxyz$

(1) if v is empty,

(i) y contains exclusively as or bs or cs.
(ii) y contains as and bs OR bs and cs.

(2) if y is empty,

(i) v contains exclusively as or bs or cs.
(ii) v contains as and bs OR bs and cs.

(3) Neither v nor y is empty

(i) v contains exclusively as or bs or cs, and
y contains exclusively as or bs or cs.

(ii) v contains exclusively as or bs or cs, and
y contains as and bs OR bs and cs.

(iii) v contains as and bs OR bs and cs, and
y contains exclusively as or bs or cs.

(iv) v contains as and bs OR bs and cs, and
y contains as and bs OR bs and cs.