CHAPTER 2: CONTEXT-FREE LANGUAGES

CONTEXT-FREE GRAMMARS ("CFGs")
   Definition & Examples
CHOMSKY NORMAL FORM
PUSHDOWN AUTOMATA
   Non-deterministic ≠ Deterministic
EQUIVALENCE BETWEEN
   CONTEXT-FREE GRAMMARS
   AND
   NON-DETERMINISTIC PUSHDOWN AUTOMATA
PROOF
PUMPING LEMMA
   TO PROVE SOME LANGUAGES
   ARE NOT CONTEXT-FREE
**Example**

\[
\begin{align*}
E & \rightarrow E + T \mid T \\
T & \rightarrow T \times F \mid F \\
F & \rightarrow (E) \mid \alpha \\
E & \rightarrow E + T \\
E & \rightarrow T \\
\end{align*}
\]

**Variables** (= "Non-Terminals")

**Terminals** (symbols from alphabet)

**Rules** (= "Productions")

\[
E ::= E + T
\]

**Start Variable**. Often "S"

Assume: First rule uses start symbol
DERIVATION

\[ E \Rightarrow E + T \Rightarrow T + T \Rightarrow E + T \Rightarrow a + T \]
\[ \Rightarrow a + F \Rightarrow a + (E) \Rightarrow a + (T) \]
\[ \Rightarrow a + (T \times F) \Rightarrow a + (F \times F) \]
\[ \Rightarrow a + (a \times a) = a + (a \times a) \]

WE WRITE:

\[ E \not\Rightarrow a + (a \times a) \]
\[ E \not\Rightarrow a + (E) \Rightarrow a + (T) \not\Rightarrow a + (a \times a) \]

LEFT-MOST DERIVATION:
Always choose Left-most Variable

\[ \ldots \Rightarrow F + T \Rightarrow a + T \Rightarrow \ldots a + (a \times a) \]

RIGHT-MOST DERIVATION:
Always choose Right-most Variable

\[ \Rightarrow F + T \Rightarrow F + F \Rightarrow \ldots a + (a \times a) \]
E *⇒ a + (a × a)

E ⇒ E + T | T
T ⇒ T × F | F
F ⇒ (E) | a
You get the same result:

\[
\begin{align*}
E & \rightarrow E + T \mid T \\
T & \rightarrow T \times F \mid F \\
F & \rightarrow (E) \mid a
\end{align*}
\]

\[\ldots \Rightarrow F + T \Rightarrow a + T \Rightarrow a + F \Rightarrow a + a\]

\[\ldots \Rightarrow F + T \Rightarrow F + F \Rightarrow F + a \Rightarrow a + a\]

Parse tree:

The parse tree abstracts away the actual order in which the rules are used. It only remembers which rules were used.
**Definition**

A "context-free grammar" ("CFG")

\[ G = (V, \Sigma, R, S) \]

\[ V = \text{The set of variables (i.e., non-terminals)} \]

\[ \Sigma = \text{The set of terminals} \]

Note: \( V \cap \Sigma = \emptyset \).

Both are finite sets.

\[ R = \text{The set of rules (i.e., productions)} \]

\[ S = \text{Start variable. } S \in V. \]

**Definition**

The language of a grammar is

\[ \{ w | w \in \Sigma^* \text{ and } S \Rightarrow w \} \]

**Definition**

A "context-free language" is a language generated by a context-free grammar.
EXAMPLE CFG

\[ S \rightarrow (S) \mid SS \mid \varepsilon \]

Language = \[ \{ \varepsilon, (), (()), ((()), (((())())(())), (((()))), (())(())(), (())(), ..., \} \]

EXAMPLE

\[ \{ 0^n 1^n \mid n \geq 0 \} \]

\[ S \rightarrow \varepsilon \]

\[ S \rightarrow 0S1 \]

We used the Pumping Lemma to show this is not regular. So:

REG C CFL

[PROPER SUBSET]
\[ E \rightarrow E + E \\
\rightarrow E \times E \\
\rightarrow (E) \\
\rightarrow a \]

\[ a + a \times a \]

- For each parse tree, there is exactly one left-most derivation. (And one right-most derivation.)

\textbf{Ambiguous Strings}
- More than one parse tree.
- Fundamentally different ways to derive this string.

\textbf{Ambiguous Grammar}
A grammar is "ambiguous" if some string can be derived in more than one way. I.e., some string has multiple parse trees.
**Ambiguous Grammar**

\[
E \rightarrow E + E \\
\rightarrow E \times E \\
\rightarrow (E) \\
\rightarrow a
\]

**Equivalent Unambiguous Grammar**

\[
E \rightarrow E + T \$
\rightarrow T \\
T \rightarrow T \times F \\
\rightarrow F \\
F \rightarrow (E) \\
\rightarrow a
\]
AMBIGUOUS STRINGS

Multiple left-most derivations = multiple parse trees.

AMBIGUOUS GRAMMAR

If there exists some string that can be derived ambiguously.

If you have an ambiguous grammar, you might be able to find an equivalent grammar that is unambiguous.

AMBIGUOUS LANGUAGE

All grammars for the language are ambiguous.
There is no unambiguous grammar that accepts the language.
The language is "inherently ambiguous."
Every regular language is context-free.

Proof:
Given a DFA for the language, construct a grammar that generates the same language.

- Make a variable for each state.
- Make the variable for the starting state the starting variable.
- Make a rule for each edge.
- Add an epsilon rule for each accept state.

\[ A \rightarrow 1B \]
\[ A \rightarrow 3C \]
\[ B \rightarrow 2B \]
\[ B \rightarrow 3D \]
\[ C \rightarrow 1D \]
\[ D \rightarrow 4D \]
\[ D \rightarrow \epsilon \]
THE LANGUAGE ONION:

All Languages

Turing Recognizable Languages
(TM will halt if "yes", but may loop if "no")

Decidable Languages
(TM will always halt)

Context-Free Languages
(Nondeterministic Pushdown Automaton)

Unambiguous Languages

LR(k) Languages
(Deterministic Pushdown Automaton)

LL(k) Languages
(Predictive Parser)

Regular Languages
(Finite State Machine, Reg. Expr.)
**CHOMSKY NORMAL FORM**

Every rule in the grammar has the form

\[ A \rightarrow BC \]

or

\[ A \rightarrow a \]

We can also have

\[ S \rightarrow \varepsilon \]

**Theorem**

Every context-free language can be generated by a grammar in Chomsky normal form.

Two grammars are "equivalent" if they generate the same language.

For every CFG there is an equivalent Chomsky normal form.
Algorithm to convert any CFG into Chomsky Normal Form

**Step 1**
Make sure start symbol does not appear on right-hand side.

**Step 2**
Remove rules like $A \rightarrow \epsilon$

**Step 3**
Get rid of all unit rules $A \rightarrow B$

**Step 4**
Get rid of rules with more than 2 symbols on right-hand side
- $A \rightarrow B C D E$
- $A \rightarrow B c d e$

**Step 5**
Make sure
- $A \rightarrow B C$
- $A \rightarrow a$

- Only 2. must be variables.
- Only 1. must be a terminal symbol.
PROOF

GIVEN A CFG $G$, SHOW HOW TO CONVERT IT TO CHOMSKY NORMAL FORM.

STEP 1: MAKE SURE START SYMBOL DOESN'T APPEAR ON RIGHHAND SIDE. ADD NEW START SYMBOL.

EXAMPLE

$S \rightarrow ASA \mid aB$
$A \rightarrow B \mid S$
$B \rightarrow b \mid \epsilon$

$S_0 \rightarrow S$

$S \rightarrow ASA \mid aB$
$A \rightarrow B \mid S$
$B \rightarrow b \mid \epsilon$
Can’t have $A \Rightarrow e$. (unless $A$ is start variable)

Remove such rules.

$$B \Rightarrow BCACBAB$$

$$B \Rightarrow BCACBAB$$

$$B \Rightarrow BCACBAB$$

$$E \Rightarrow BCACBB$$

$$B \Rightarrow BCBB$$

Add these new rules.

---

$$S_0 \Rightarrow S$$
$$S \Rightarrow ASA | aB$$
$$A \Rightarrow B | S$$
$$B \Rightarrow b | e$$

Remove $B \Rightarrow e$

$$S_0 \Rightarrow S$$
$$S \Rightarrow ASA | aB | a$$
$$A \Rightarrow B | S | e$$
$$B \Rightarrow b$$

Remove $A \Rightarrow e$

$$S_0 \Rightarrow S$$
$$S \Rightarrow ASA | aB | a$$
$$A \Rightarrow B | S$$
$$B \Rightarrow b$$
**STEP 3:**

GET RID OF ALL "UNIT RULES"

\[ A \rightarrow B \]

**GIVEN:**

\[ B \rightarrow xyz \]

**ADD:**

\[ A \rightarrow xyz \]

**FROM BEFORE:**

\[ S_0 \rightarrow S \]
\[ S \rightarrow ASA | ab | a | SA | AS | S \]
\[ A \rightarrow B | S \]
\[ B \rightarrow b \]

Remove \[ S \rightarrow S \]

(Do nothing)

\[ S_0 \rightarrow S \]
\[ S \rightarrow ASA | ab | a | SA | AS \]
\[ A \rightarrow B | S \]
\[ B \rightarrow b \]

Remove \[ S_0 \rightarrow S \]

\[ S_0 \rightarrow ASA | ab | a | SA | AS \]
\[ S \rightarrow ASA | ab | a | SA | AS \]
\[ A \rightarrow B | S \]
\[ B \rightarrow b \]
FROM BEFORE:

\[ S_0 \rightarrow ASA \mid aB \mid a \mid SA \mid AS \]
\[ S \rightarrow ASA \mid aB \mid a \mid SA \mid AS \]
\[ A \rightarrow B \mid S \]
\[ B \rightarrow b \]

REMOVE \( A \rightarrow B \):

\[ S_0 \rightarrow ASA \mid aB \mid a \mid SA \mid AS \]
\[ S \rightarrow ASA \mid aB \mid a \mid SA \mid AS \]
\[ A \rightarrow b \mid S \]
\[ B \rightarrow b \]

REMOVE \( A \rightarrow S \):

\[ S_0 \rightarrow ASA \mid aB \mid a \mid SA \mid AS \]
\[ S \rightarrow ASA \mid aB \mid a \mid SA \mid AS \]
\[ A \rightarrow b \mid ASA \mid aB \mid a \mid SA \mid AS \]
\[ B \rightarrow b \]
STEP 4:
Replace
\[ A \rightarrow BCDE \]
with
\[ A \rightarrow BA, \]
\[ A_1 \rightarrow CA_2 \]
\[ A_2 \rightarrow D \]
\[ E \]
Introduce new variables to make sure every right-hand side is not longer than 2.

FROM BEFORE:
\[ S_0 \rightarrow ASA \mid aB \mid a \mid SA \mid AS \]
\[ S \rightarrow ASA \mid aB \mid a \mid SA \mid AS \]
\[ A \rightarrow b \mid ASA \mid aB \mid a \mid SA \mid AS \]
\[ B \rightarrow b \]

SHORTEN \[ S_0 \rightarrow ASA \] and \[ S \rightarrow ASA \] and
By introducing \[ A_1 \]:
\[ A \rightarrow ASA \]
\[ S_0 \rightarrow AA_1 \mid aB \mid a \mid SA \mid AS \]
\[ S \rightarrow AA_1 \mid aB \mid a \mid SA \mid AS \]
\[ A \rightarrow b \mid AA_1 \mid aB \mid a \mid SA \mid AS \]
\[ A_1 \rightarrow SA \]
\[ B \rightarrow b \]
Step 5:

Replace $A \rightarrow b C$

with:

$A \rightarrow A_1 C$

$A_1 \rightarrow b$

From Before:

$S_0 \rightarrow AA_1 | aB | a | SA | AS$

$S \rightarrow AA_1 | aB | a | SA | AS$

$A \rightarrow b | AA_1 | aB | a | SA | AS$

$A_1 \rightarrow SA$

$B \rightarrow b$

INTRODUCE $A_2 \rightarrow a$

$S_0 \rightarrow AA_1 | A_2 B | a | SA | AS$

$S \rightarrow AA_1 | A_2 B | a | SA | AS$

$A \rightarrow b | AA_1 | A_2 B | a | SA | AS$

$A_1 \rightarrow SA$

$B \rightarrow b$

$A_2 \rightarrow a$

This grammar is now in Chomsky normal form!
INPUT STRING:
SAME AS F.S.M.
(CANNOT BACK UP)

STACK
OPERATIONS:
READ+POP / IGNORE
PUSH / IGNORE
STACK ALPHABET
Γ = GAMMA,
MAY BE DIFFERENT FROM INPUT
ALPHABET, Σ

STATE TRANSITIONS:
MAY DEPEND ON STACK TOP.
MAY PUSH ONTO THE STACK.
NON-DETERMINISTIC!
Finite State Machine

Pushdown Automaton

Input symbol

Symbol on top of the stack. This symbol is popped.

"ε" means the stack is neither read nor popped

Non-determinism

This symbol is pushed onto the stack.

"ε" means nothing is pushed.
\[ \{0^n1^n \mid n \geq 0\} \]

\[ \Sigma = \{0, 1\} \quad \text{input alphabet} \]

\[ \Gamma = \{\$, 0\} \quad \text{stack alphabet} \]

To detect bottom of stack.
(We could use other symbols, but these seem clear enough.)
WHEN IS A STRING ACCEPTED?

- Begin in the start state.
- End in an accept state.
- Consume all the input symbols.
  (Okay to leave stuff on the stack)
- There is a path thru the finite state control.

**Note:** It is not possible to pop an empty stack.

- Non-deterministic:
  You just have to find one path to an accept state.

\[
\begin{align*}
  a, b &\Rightarrow c \\
  a, \epsilon &\Rightarrow c
\end{align*}
\]

- Read a "b" and pop it.
- Don't look at the stack.
Formal Definition

\[(Q, \Sigma, \Gamma, S, q_0, F)\]

\[Q = \text{Set of states}\]
\[\Sigma = \text{Input alphabet}\]
\[\Gamma = \text{Stack alphabet}\]

\[\Sigma_e = \Sigma \cup \{\varepsilon\}\]
\[\Gamma_e = \Gamma \cup \{\varepsilon\}\]

\[S: Q \times \Sigma_e \times \Gamma_e \rightarrow \mathcal{P}(Q \times \Gamma_e)\]

\[q_0: \text{Starting State} \quad q_0 \in Q\]

\[F: \text{Accept States} \quad F \subseteq Q\]
EXAMPLE

Palindrome

MADAM IM ADAM
WAS IT A CAT I SAW
NO LEMON, NO MELON

\[ \exists w \mid w \text{ is a palindrome AND } w \in \{0, 1\}^* \]

\[
\begin{align*}
S & \rightarrow 0S0 \\
& \rightarrow 1S1 \\
& \rightarrow \epsilon
\end{align*}
\]

\[
\begin{align*}
A & \rightarrow \epsilon, \epsilon \rightarrow \$
B & \rightarrow \epsilon, \epsilon \rightarrow \epsilon \\
C & \rightarrow \epsilon, \epsilon \rightarrow \epsilon \\
D & \rightarrow \epsilon, 0 \rightarrow \epsilon \\
& \rightarrow \epsilon, 1 \rightarrow \epsilon
\end{align*}
\]
GRAMMAR DESIGN CHALLENGE

\[ L = \{w \mid w \in \{0,1\}^* \text{ and the number of } 0\text{'s equals the number of } 1\text{'s}\} \]

**Approach:** Try to think of "meanings" for the non-terminals.

- **S** = equal # of 0's and 1's.
- **A** = one more "1" than "0"'s.
- **B** = one more "0" than "1"'s.

**Solution:**

\[ S \rightarrow OA \mid 1B \mid \epsilon \]
\[ A \rightarrow 1S \mid OAA \]
\[ B \rightarrow 0S \mid 1BB \]
SOLUTION #2:

\[ S \rightarrow S A B \mid \epsilon \]
\[ A \rightarrow 0 5 1 \mid \epsilon \]
\[ B \rightarrow 1 5 0 \mid \epsilon \]

**Note:**

\[ C \rightarrow C \times \mid \epsilon \]

Generates: \( \times^* \)

\( \{ \epsilon, x, xx, xxx, xxxx, \ldots \} \)

\[ S \rightarrow S A B \mid \epsilon \]

Generates:

\( A B A B A B A B \ldots \)

Every \( A \) can go to \( \epsilon \).
Every \( B \) can go to \( \epsilon \).

\( \Rightarrow \) Any string of \( A \)'s and \( B \)'s!

0 1 0 1 0 1 0 1
A A A A A A

0 0 0 0 1 1 1 1 1
S

A
ARE TWO GRAMMARS EQUIVALENT?

"EQUIVALENT" = GENERATE THE SAME LANGUAGE

UNDECIDABLE!

CANNOT WRITE A COMPUTER PROGRAM.
  [PROGRAM MAY NOT HALT!]

APPROACH:

• GENERATE EVERY STRING IN TURN. (INFINITELY MANY).
• TEST EACH STRING.
  ACCEPTED BY GRAMMAR #1?
  FIND A PARSE TREE.
  (THIS IS DECIDABLE.)
  ACCEPTED BY GRAMMAR #2?
• FIND A COUNTER-EXAMPLE?
  HALT; PRINT "NOT EQUIVALENT!"
• OTHERWISE, KEEP LOOKING.
  MAY NOT HALT... OR MAY...
• NO WAY TO KNOW WHEN TO STOP LOOKING!!!

**Non-context-free Grammars**

\[ L = \{ a^n b^n | n \geq 0 \} \]

Is this language context-free?

No!

(Use the Pumping Lemma for CFG's to prove it.)

---

**Chomsky Hierarchy**

- **Type-3 Languages** (Regular)
- **Type-2 Languages** (Context-Free)
- **Type-1 Languages** (Context-Sensitive)
- **Type-0 Languages** (Recursively Enumerable, Turing-Enumerable, Turing-Recognizable)

Example: \[ ABx \rightarrow ADEy \]
**Approach:**

Each "A" will turn into a "1".
Each "B" will turn into a "2".
Each "C" will turn into a "3".

\[
S \xrightarrow{*} \ldots \text{AAA BBB CCC} \xrightarrow{*} \ldots \text{111 222 333}
\]

Initially the string will start with "1"s.

111 -> BBB CCC

Rules to finish it up:

1B → 12
2B → 22
2C → 23
3C → 33

Note

... CB.... Can never be reduced.
"C" can only turn into a "3".
And "3B" can never be reduced.
⇒ B's must precede "C"'s.
STEP 1: GENERATE THE CORRECT NUMBER OF S's, B's, and C's!
(JUST NOT THE RIGHT ORDER.)

\[ S \rightarrow 1SBC \]
\[ S \rightarrow e \]
\[ 1111BCBCBCBC \]

STEP 2: GET THE "B"S IN FRONT OF THE "C"S.

\[ CB \rightarrow eBC \]

STEP 3:
REDUCE "B" TO "2" AND "C" TO "3."
(Rules shown above.)
CONTEXT-SENSITIVE GRAMMAR

\[ L = \{ 1^N 2^N 3^N \mid N \geq 0 \} \]

\[
\begin{align*}
S & \rightarrow 1SBC \\
S & \rightarrow \varepsilon \\
CB & \rightarrow HB \\
HB & \rightarrow HC \\
HC & \rightarrow BC \\
1B & \rightarrow 12 \\
2B & \rightarrow 22 \\
2C & \rightarrow 23 \\
3C & \rightarrow 33
\end{align*}
\]
Are Context-Free Languages Closed Under Union?

Grammar 1:
\[ S_1 \rightarrow \text{... lots of rules...} \]

Grammar 2:
\[ S_2 \rightarrow \text{... lots of rules...} \]

Union:
\[ S \rightarrow S_1 \mid S_2 \]
\[ S_1 \rightarrow \text{...} \]
\[ S_2 \rightarrow \text{...} \]

Yes!

Are Context-Free Languages Closed Under Concatenation?

\[ S \rightarrow S_1 \cdot S_2 \]
\[ S_1 \rightarrow \text{...} \]
\[ S_2 \rightarrow \text{...} \]

Yes!
Are context-free languages closed under intersection?

Consider $L_1 = \{0^n1^2\mid n \geq 0\}$

It is a "CFL".

- $S \rightarrow AB$
- $A \rightarrow 0A1 \mid \varepsilon$
- $B \rightarrow 2B \mid \varepsilon$

Consider $L_2 = \{0^k1^n2^m\mid n \geq 0, m \geq 0\}$

It is a "CFL".

- $S \rightarrow AB$
- $A \rightarrow 0A \mid \varepsilon$
- $B \rightarrow 1B2 \mid \varepsilon$

Consider $L = L_1 \cap L_2$

$\{0^n1^2\mid n \geq 0\}$

This is not A context-free language.

No!

Not in general, but for some languages the intersection may also be context-free.
Are context-free languages closed under complement?

These are sets.
Demorgan's laws apply.

\[ A \cap B = \overline{A} \cup \overline{B} \]

Assume: closed under complement.
Then right-hand side is closed.
Then left-hand side is a C.F.L.

Contradiction.

No! (Not in general, but yes for some languages)

Example

\[ L = \{ \text{ww} \mid w \in \{0,1\}^* \} \]

The first half of the string is equal to the second half.

\( L \) is not context-free.

However

\( \overline{L} \) is context-free.
**THEOREM**

A LANGUAGE IS CONTEXT-FREE IFF SOME PUSHDOWN AUTOMATON RECOGNIZES IT.

**PROOF**

**PART 1:**

Given a CFG, show how to construct a pushdown automaton that recognizes it.

**PART 2:**

Given a pushdown automaton, show how to construct a context-free grammar that recognizes the same strings.
Given: A grammar

\[
S \rightarrow BS | A \\
A \epsilon \rightarrow 0A | \epsilon \\
B \rightarrow BB1 | 2
\]

Find: (or build) a PDA.

Consider a derivation: (left-most)

\[
S \xrightarrow{1} BS \\
\xrightarrow{2} BB1S \\
\xrightarrow{3} 2B1S \\
\xrightarrow{4} 221S \\
\xrightarrow{5} 221A \\
\xrightarrow{6} 221
\]

General form:

\[
\text{Terminals: } aaaaaaBbBbAa \\
\text{Rest (both): } \text{Terminals}
\]
INPUT:

[Diagram showing a stack and input sequence]

LEFT-MOST DERIVATION:

S \Rightarrow \ldots aaaaa BaBC \Rightarrow \ldots

AT EACH STEP, EXPAND LEFT-MOST NON-TERMINAL.

Rule:

B \Rightarrow ASA \times BA

\ldots \Rightarrow aaaaa ASA \times BA aBC

So:

- MATCH STACK-TOP TO A RULE
- POP STACK
- PUSH RIGHT-HAND SIDE OF RULE ONTO STACK.
Rule: \( A \rightarrow BCD \)

Add this to PDA

\[ \epsilon, A \rightarrow BCD \]

Right-hand side.

Match Top and Pop.

Input is not advanced.

Note:
To push multiple items, you'll need to add some extra states.

\[ \epsilon, A \rightarrow D \rightarrow \epsilon, \epsilon \rightarrow C \rightarrow \epsilon, \epsilon \rightarrow B \]

Which rule to use?

P.D.A.'s are non-deterministic!

(Try them all in parallel.)

(Just choose the "right" rule.)
Rule:

\[ A \rightarrow 0102B3C \]

Gives this:

PDA

INPUT

...40102...

So:

MATCH TERMINAL SYMBOLS TO THE STACK TOP.

\[
\begin{align*}
0,0 & \rightarrow \epsilon \\
1,1 & \rightarrow \epsilon \\
2,2 & \rightarrow \epsilon \\
& \text{etc. for all } x \in \mathbb{Z}.
\end{align*}
\]
THE FINAL MACHINE:

\[ e, e \rightarrow \$ \]

\[ e, e \rightarrow \$ \]

\[ e, a \rightarrow e \]

\[ x, x \rightarrow \$ \text{ for } x \in \Sigma \]

\[ e, A \rightarrow BCD \]

for all rules \[ A \rightarrow BCD. \]
Proof, Part 2

We are given: a P.D.A.
Must build: a CFG from it.

Step 1
Simplify the PDA.

Step 2
Build the CFG.

There will be a non-terminal for every pair of states.
\( A_p \ A_q \ A_{qr} \ A_{q0} \ldots \)
The starting non-terminal will be
\( A_{00} \ A_{q0} \ A_{q0} \text{ACCEPT} \)
**Simplify the PDA**

1. The PDA has only one ACCEPT state.

2. The PDA empties its stack before accepting.

\[ \epsilon, \emptyset \rightarrow \epsilon \]

\[ \epsilon, x \rightarrow \epsilon \] for all \( x \in \Gamma - \{\#\} \)
③ Each transition either pushes or pops, but does not do both.

\[ a, X \rightarrow Y \]

\[ a, X \rightarrow \varepsilon \]

\[ \varepsilon, \varepsilon \rightarrow Y \]

\[ a, \varepsilon \rightarrow \varepsilon \]

\[ \varepsilon, \varepsilon \rightarrow \varepsilon \]

\[ a, \varepsilon \rightarrow Z \]

\[ \varepsilon, Z \rightarrow \varepsilon \]

\[ \varepsilon \rightarrow \text{Dummy} \]

Add some new symbol to stack alphabet.
"Don't modify the stack"

= "Start w/ an empty stack and finish with an empty stack"

= "Don't touch the stack"

---

Diagram of stack operations:

- Push operation shown with elements a, b, c
- Pop operation shown with elements a, b, c
- Stack size chart showing fluctuations

- First thing is always "push"
- Last thing is always a "pop"
- Nothing here is touched during the computation
Consider two states $p$ and $q$ in the PDA.

Could we go from $p$ to $q$ without touching the stack?

What strings would do that?

That is:
Starting w/ an empty stack,
we could go from $p$ to $q$
and end up with an
empty stack.

Or if somethings were on the
stack they would never be touched.

The grammar we build will have
a non-terminal

$\Rightarrow$ we'll call it $A_{pq}$

that will generate exactly these
strings!
What strings can be generated/accepted by following this path?

"a ... b"

This rule will generate exactly those strings!!
What strings can be generated by following this path?

```
"aaaa...b...bb"
```

From 0 to R, From R to 9

This rule will generate exactly those strings!
If we have these edges

\[ a, \varepsilon \rightarrow t \]

\[ \text{PUSH} \]

\[ b, t \rightarrow \varepsilon \]

\[ \text{POP} \]

And we could get from \(\textcircled{0}\) to \(\textcircled{5}\) without touching the stack,

Then we need a new grammar rule:

\[ A_{pq} \rightarrow aA_{rs}b \]

For each \(p, q, r, s \in EQ\) in the PDA, such that \(S(p, a, \varepsilon) = \) contains \((r, t)\),

and \(S(s, b, t) = \) contains \((q, \varepsilon)\)

[i.e., "and the edges are labelled as above, for any \(a, b \in \Sigma\) and \(t \in \Gamma\)..."

Then add this rule to the CFG:

\[ A_{pq} \rightarrow aA_{rs}b \]
If we have a way to get from state (p) to state (r) that doesn't touch the stack.

\[ p \rightarrow r \]

And a way to get from (r) to (q) that doesn't touch the stack.

\[ r \rightarrow q \]

Then we have a new way to get from (p) to (q) without touching the stack.

For every state \( p, r, q \in Q \), add this rule to the grammar:

\[ Apq \rightarrow Apr Arg \]
There is a trivial way to get from state $p$ to itself without touching the stack: The string $\epsilon$. So add

$$A_{pp} \rightarrow \epsilon \quad \text{for every state } p.$$

If the PDA accepts some string then there is a way to go from $q_0$ to $q_{\text{Accept}}$ that does not modify the stack.

The grammar we seek should accept generate exactly these strings.

Our START NON-TERMINAL is:

$$A_{q_0q_{\text{Accept}}}$$
PUMPING LEMMA FOR CFG'S

\[ s \]

\[ \text{abbaba} \]

\[ \text{really long string} \]

Some non-terminal "R" must be used more than once.
Consider: \( L = \{0^n1^n2^n3^n4^4 | n \geq 0\} \)

\[ S \rightarrow 00R44 \]
\[ R \rightarrow 1R34 \]

How can we generate "really long" strings?
\[ S \Rightarrow 00R44 \]
\[ R \Rightarrow 1R31z \]

\[ \begin{array}{c}
00 & 44 \\
\hline
u & x & z \\
00 & 123 & 44 \\
\hline
u & v & x & y & z \\
00 & 1111 & 2333 & 344 \\
\hline
u & v^4 & x & y^4 & z \\
\end{array} \]

\[ uv^i x y^i z \]

is also in
the Language!
For all strings that are "long enough," some non-terminal has to be repeated in the parse tree.

Therefore, these are also legal parse trees:

\[ R \Rightarrow \text{UV}xyz \]

is also in the language!
In other words, to get long strings we must use recursion in the grammar.

\[ R \Rightarrow \ldots R \ldots \]

And finally to finish:

\[ R \Rightarrow x \]
PUMPING LEMMA FOR CFG's

If a string is sufficiently long, $|s| \geq p$ then it can be pumped. That is, the string can be broken into parts (some way)

$s = uvxyz$

such that all strings of the form $uv^ixy^iz$ are also in the language.

NOTES:

- $v$ and $y$ cannot both be $\epsilon$
  $|vy| > 0$

- The beginning of $v$ and the end of $y$ can't be too far apart
  $|vxy| \leq p$
PUMPING LEMMA FOR CFG's (REWORDED IT)

If A is a context-free language, then there is a pumping length p such that, for any string in A whose length is long enough, |s| ≥ p, that string can be broken into pieces \( s = uvxyz \) in a way that satisfies all three of these conditions:

**Condition 1:**
\( uv^ixyz \) is in A, for all \( i ≥ 0 \)

**Condition 2:**
\( |vy| > 0 \)

**Condition 3:**
\( |vxy| ≤ p \)
Logic Refresher

How can we negate

"For all"

\[ \sim \forall x. p(...) \equiv \exists x. \sim p(...) \]

"There exists"

\[ \sim \exists x. p(...) \equiv \forall x. \sim p(...) \]

"And"

\[ \sim (\ldots p_1 \land \ldots q_{\ldots}) \equiv (\forall (\ldots p_1 \ldots)) \lor (\sim (\ldots q_{\ldots})) \]

"It's not the case that all numbers are even."

"There exists a number that is not even."

"It's not the case that there exists a green number."

"All numbers have the "not-green" property."
PUMPING LEMMA LOGIC

If \( L \) is a context-free language ....

**PUMPING PROPERTY**

\[ \exists p \]
\[ \forall s \text{ in } L \text{ where } |s| \geq p \]
\[ \exists uv^iwxz = s \]
Such that
1. \( uv^ixyz \in L, \forall i \geq 0 \) AND
2. \( |vy| > 0 \) AND
3. \( |vxy| \leq p \)

To show \( L \) is not context-free, we must show that \( \sim \) (pumping property) holds.

**NOT-PUMPING PROPERTY**

\[ \forall p \]
\[ \exists s \text{ in } L \text{ where } |s| \geq p \]
\[ \forall uv^ixyz = s \]
Such that
1. \( uv^ixyz \notin L, \forall i \geq 0 \) OR
2. \( |vy| \neq 0 \) OR
3. \( |vxy| \neq p \)
Show \( B = \{ a^n b^m c^n \mid n \geq 0 \} \) is not context-free.

Assume it is CFL. Show \( \sim \) (pumping property)

Let \( p \) be the pumping length.

(No constraints on \( p \). We'll show it \( \not\in \).

There exists a string... \( s \in \mathbb{L} \)

We'll use: \( a^p b^p c^p \) \([\exists s...\])

Now look at all ways to divide it up.

\([ \forall uvxyz = s...\])

We'll show some condition will always be violated.

Condition (2) says \( |vy| > 0 \)

Assume this condition \( \% \) holds.

Now look at two cases.

- \( \neg \text{cond 1} \) or
- \( \neg \text{cond 2} \) or
- \( \neg \text{cond 3} \)

- \( \neg \text{cond 2} \)
- \( \text{cond 2} \)
- \( \text{show } \neg \text{cond 1} \)
**CASE 1**

V and \( y \) each contain only one type of symbol.

\[
\begin{align*}
\text{a a a a b b b b c c c c} & \\
\text{v} & \\
\text{y} & \\
\text{a a a a b b b b c c c c} & \\
\text{v} & \\
y = \varepsilon
\end{align*}
\]

One symbol will always be left out.

Pump \( s \) to \( uv^2xy^2z \)

\[
\begin{align*}
\text{a a a a a a b b b b c c c c} & \\
\text{v} & \\
\text{v} & \\
\text{y} & \\
y & \\
y
\end{align*}
\]

At least one symbol will increase in number.

At least one symbol will not increase in number.

The string cannot still be in the form

\[
a^n b^n c^n
\]
CASE 2

Either \( v \) or \( y \) has more than one kind of symbol:

\[
\begin{array}{c}
\text{a a a b b b c c c} \\
\text{\underline{v} y}
\end{array}
\]

\[
\begin{array}{c}
\text{a a a b b b c c c} \\
\text{\underline{v} y}
\end{array}
\]

Pump to \( UV^2xy^2z \). We might have right number of symbols, but the order will be wrong.

\[
\begin{array}{c}
\text{a a a a b b b c b c c c} \\
\text{\underline{v v} \underline{y y}}
\end{array}
\]
Show that $D = \{ w w \mid w \in \{0,1\}^* \}$ is not context-free.

Assume it is a CFL.

Show \textit{~(Pumping Property)}

Let $p$ be the pumping length.

[No constraints on $p$. Show $\forall p$.]

There exists a string $s$, $|s| \geq p$...

[To show $\exists s$, just provide an example; just give a $s$ that exists.]

$0^p1^p0^p1^p$

Look at all ways to divide $s$ into parts.

[Show $\forall uvxyz = s$]

Must show that some condition is not satisfied.

We'll assume condition 3 holds

$|vxy| \leq p$

and show that the other conditions must fail.
Consider the boundaries between 0's and 1's in our string.

\[ \text{CASE 1: } vxy \text{ does not straddle a boundary.} \]

Pumping up will yield a string with more 1's and an "imbalance."

The string now has the form

This string is not of the form \( W^W \). Since \( UV^2XV^2Z \) is not in the language, condition 1 is violated.
CASE 2: \(vxy\) straddles the first boundary.

\[
00000\{11111\{00000\}11111
\]

\(vxy\) \(\rightarrow\) MIDPOINT.

Since \(|vxy| \leq p\) it cannot straddle the midpoint.

PUMPING DOWN WILL MAKE THE STRING SHORTER.

\[
000 \overline{1100000,11111}
\]

FEWER SYMBOLS

NEW MIDPOINT

\[
000 \overline{1100000,11111}
\]

P P

NOTE: The string now has the form:

NEW MIDPOINT.

\[
\overline{0-------------1}
\]

The string is not of the form \(WW\).

CASE 2b: \(vxy\) straddles the third boundary: It is similar.

\[
00000\{11111\{00000\}11111
\]

\(vxy\) \(\rightarrow\)
CASE 3: \( vxy \) straddles the midpoint.

Since \( |vxy| \leq p \), it cannot also straddle the first or third boundary.

Pumping down will give us a shorter string.

Look at the first half of the string and the second half.

They cannot be equal. This string fails condition 1.